Table of Contents

一些使用Swift的iOS教程

> - Gitbook
> - Github

目录

30. Xcode 8中使用Swift 3构建条形码与QR码识别器
31. Core Plot指南:入门
32. 奇妙的Swift错误处理
33. (进行中)UIPresentationController指南:入门
34. (未开始)RxSwift与RxCocoa入门
35. (未开始)iOS Accessibility指南:入门

来源

- Ray Wenderlich
- AppCoda
- iOS-Blog
- Jameson Quave

译者

- yrq110
- bugcode

> 人手有限，若有翻译不当之处可直接PR或发送邮件到yrq1025@gmail.com即可。

Swift Tutorial: Working with JSON

Swift指南：使用JSON

> - 原文链接：Swift Tutorial: Working with JSON
> - 原文作者：Attila Hegedüs
> - 译者：yrq110

> 更新至Xcode 7.1与Swift 2 12-21-2015

JavaScript Object Notation或简称 `JSON` ，是一种与Web端服务进行数据传递的通用方法。它具有易用与可读性强的特点，这也是它为何会这么火的原因。

考虑一下下面这段JSON：

```
[
  {
    "person": {
      "name": "Dani",
      "age": "24"
    }
  },
  {
    "person": {
      "name": "ray",
      "age": "70"
    }
  }
]
```

在 `Objective-C` 中，对JSON的解析与反序列化是很明确的：

```
NSArray *json = [NSJSONSerialization JSONObjectWithData:JSONData options:kNilOptions e
rror:nil];
NSString *age = json[0][@"person"][@"age"];
NSLog(@"Dani's age is %@", age);
```

在 `Swift` 中，由于Swift的可选类型与类型安全，同样的操作显得更加繁琐：

```
var json: Array!
do {
  json = try NSJSONSerialization.JSONObjectWithData(JSONData, options: NSJSONReadingOp
tions()) as? Array
} catch {
  print(error)
}

if let item = json[0] as? [String: AnyObject] {
  if let person = item["person"] as? [String: AnyObject] {
    if let age = person["age"] as? Int {
      print("Dani's age is \(age)")
    }
  }
}
```

在上面的代码中，当你使用JSON中的每一个object之前都需要通过 `可选绑定` (optional binding)进行检查。这会保证你代码的安全性，但是若你的JSON越复杂，你的代码就会变得越丑。

使用Swift2.0中的 `guard` 语句可以帮助避免if语句的嵌套：

```
guard let item = json[0] as? [String: AnyObject],
  let person = item["person"] as? [String: AnyObject],
  let age = person["age"] as? Int else {
    return;
}
print("Dani's age is \(age)")
```

还是太冗长，不是吗？你怎样才能使它更简洁？

在这篇JSON教程中，你会学到一种更轻松的解析JSON的方法——使用开源库Gloss

你将会使用Gloss去解析一个包含US App商店Top25应用信息的JSON文档。你将会发现在Objective-C中也能轻易的使用！

目录

- 检索远程JSON
- Gloss的工作原理

入门

下载这篇教程的开始playground

由于用户界面在这篇JSON教程中并不是很重要，因此你将只使用playgrounds进行学习。

在Xcode中打开Swift.playground看一看。

> 你可能会发现playground工程的导航默认是关闭的。使用`command`+`1`来显示它。

在开始playground文件中有一些source与resource文件，会使你更加专注于使用Swift进行JSON的解析。看一看它的结构与概要：

- `Resources`文件夹绑定了你的Swift代码中要访问的资源文件
    - `topapps.json`：包含需要解析的JSON字符串
- `Sources`文件夹包含你的主playground可以访问的其他Swift源文件，在Sources文件夹中添加额外的.swift文件辅助会使你的playground变得更加简洁与易读。
    - `App.swift`：一个原始的swift结构体，代表一个app。你的目标就是解析JSON转换为这些对象的一个集合。
    - `DataManager.swift`：管理来自本地或网络的数据检索，之后你将会使用这个文件中的方法加载一些JSON。

当你感到对当前的playground充分理解后请继续向下阅读！

用Swift原生方法解析JSON

首先，你需要从使用Swift原生方法解析JSON开始，不使用任何的外部库。这将会帮助你理解使用像Gloss这样的库的益处。

> 若你已经深知使用原生方法解析JSON的痛处，请直接跳到下一节

来让我们从提供的JSON文件中提取出AppStore中排名第一App的名字吧！

首先，当准备大量使用字典时，在playground的头部定义一个类型别名：

```
typealias Payload = [String: AnyObject]
```

如下面所示完善回调函数getTopAppsDataFromFileWithSuccess中的代码：

```
DataManager.getTopAppsDataFromFileWithSuccess { (data) -> Void in

  var json: Payload!

  // 1
  do {
    json = try NSJSONSerialization.JSONObjectWithData(data, options: NSJSONReadingOpti
ons()) as? Payload
  } catch {
    print(error)
    XCPlaygroundPage.currentPage.finishExecution()
  }

  // 2
  guard let feed = json["feed"] as? Payload,
    let apps = feed["entry"] as? [AnyObject],
    let app = apps.first as? Payload
    else { XCPlaygroundPage.currentPage.finishExecution() }

  guard let container = app["im:name"] as? Payload,
    let name = container["label"] as? String
    else { XCPlaygroundPage.currentPage.finishExecution() }

  guard let id = app["id"] as? Payload,
    let link = id["label"] as? String
    else { XCPlaygroundPage.currentPage.finishExecution() }

  // 3
  let entry = App(name: name, link: link)
  print(entry)

  XCPlaygroundPage.currentPage.finishExecution()

}
```

上面发生了什么：

1. 首先使用 `NSJSONSerialization` 对数据进行了反序列化
2. 需要逐一检查JSON对象中的所有值确保他们不为空(nil)。一旦确定它含有一个有效的值则继续检查下一个object。当使用这种方法遍历了所有下标则会得到所需要的name与link值。
3. 最后一步就是使用name与link值初始化一个App对象并使用控制台输出。

保存并运行stroyboard，应该会在调制窗口中有如下的结果：

> App(name: "Game of War - Fire Age", link: "https://itunes.apple.com/us/app/game-of-war-fire-age/id667728512?mt=8&uo=2")

是的。“Game of War – Fire Age”就是JSON文件中的第一个app

仅仅为了检索出第一个app的名字就用了这么冗长的代码，是时候看看Gloss的本事了。

JSON对象映射介绍

对象映射是一个将JSON对象转换为本地Swift对象的一种技术。在定义后模型对象与响应的映射规则后，Gloss就会代替你承担解析的重担。

这种方法明显优于你刚刚尝试的方法：

- 代码会变得更加简洁、可复用并且容易维护
- 可以操作对象而不是数组与字典
- 可以扩展模型类，添加一些额外功能

听起来不错？让我们来看看它是怎样工作的！

使用Gloss解析JSON

为了干净工整的代码环境，新建一个名为Gloss.playground的playground，然后将topapps.json复制进Resources中，将DataManager.swift复制进Soureces中。

在你的工程中集成Gloss

在工程/playground中集成Gloss：

1. 点击下面的地址Gloss Repo Zip File然后保存到一个熟悉的地方。
2. 解压后将Gloss-master/Gloss/Gloss文件夹拖进你的Sources文件夹中。

你的工程导航栏应该像下面一样：

成功了！现在你将Gloss添加进palyground了，可以不用使用令人头疼的可选绑定来解析JSON了！

> 也可以使用Cocoapods来安装Gloss。由于playground已经不再支持，只能在标准的Xcode工程中使用这种方法

映射JSON到对象

首先，需要根据你的JSON文档定义自己的模型对象

模型对象必须遵守 `Decodable` 协议，会使他们能。为了实现它，需要在协议中实施 `init?(json: JSON)` 进行初始化

观察一下topapps.json的结构来创建数据模型吧！

TopApps

TopApps模型代表最顶层的对象，它包含一个键值对：

```
{
  "feed": {
    ...
  }
}
```

在playground的sources文件夹下创建一个名为 `TopApps.swift` 的swift文件并添加如下代码：

```
public struct TopApps: Decodable {

  // 1
  public let feed: Feed?

  // 2
  public init?(json: JSON) {
    feed = "feed" <~~ json
  }

}
```

1. 定义你模型的属性。在这里仅仅只有一个。接下来你会定义 `Feed` 模型对象
2. 确保 `TopApps` 遵守 `Decodable` 协议并进行了自定义初始化

你也许想知道<~~是个啥。它叫做编码操作符(Encode Operator)，在Gloss中的Operators.swift文件中所定义。它会告诉Gloss抓取属于key键的值或对于进行编码。 `Feed` 是一个 `Decodable` 对象，因此Gloss可以将编码的任务委托给这个类。

Feed

`Feed` 对象与顶层的对象非常相似。 `Feed` 有两对键值。你只对top25的app感兴趣，因此不需要去处理author对象。

```
{
  "author": {
    ...
  },
  "entry": [
    ...
  ]
}
```

在playground的 `sources` 文件夹下创建一个名为 `Feed.swift` 的swift文件并添加如下代码：

```
public struct Feed: Decodable {

  public let entries: [App]?

  public init?(json: JSON) {
    entries = "entry" <~~ json
  }

}
```

App

`App` 是定义的最后一个模型对象。它代表表中的一个app：

```
{
  "im:name": {
    "label": "Game of War - Fire Age"
  },
  "id": {
    "label": "https://itunes.apple.com/us/app/game-of-war-fire-age/id667728512?mt=8&uo
=2",
    ...
  },
  ...
}
```

在playground的sources文件夹下创建一个名为 `App.swift` 的swift文件并添加如下代码：

```
public struct App: Decodable {

  // 1
  public let name: String
  public let link: String

  public init?(json: JSON) {
    // 2
    guard let container: JSON = "im:name" <~~ json,
      let id: JSON = "id" <~~ json
      else { return nil }

    guard let name: String = "label" <~~ container,
      link: String = "label" <~~ id
      else { return nil }

    self.name = name
    self.link = link
  }

}
```

1. `Feed` 与 `Topapp` 都使用的可选属性。如果你确信JSON中总有填充值的话可以定义一个非可选属性。
2. 不需要为JSON中的每一个成员都创建模型对象。

现在模型类已经准备好了，让Gloss工作吧！

打开playground文件并添加如下代码：

```
import UIKit
import XCPlayground

XCPlaygroundPage.currentPage.needsIndefiniteExecution = true

DataManager.getTopAppsDataFromFileWithSuccess { (data) -> Void in
  var json: [String: AnyObject]!

  // 1
  do {
    json = try NSJSONSerialization.JSONObjectWithData(data, options: NSJSONReadingOpti
ons()) as? [String: AnyObject]
  } catch {
    print(error)
    XCPlaygroundPage.currentPage.finishExecution()
  }

  // 2
  guard let topApps = TopApps(json: json) else {
    print("Error initializing object")
    XCPlaygroundPage.currentPage.finishExecution()
  }

  // 3
  guard let firstItem = topApps.feed?.entries?.first else {
    print("No such item")
    XCPlaygroundPage.currentPage.finishExecution()
  }

  // 4
  print(firstItem)

  XCPlaygroundPage.currentPage.finishExecution()

}
```

1. 首先对数据进行了反序列化操作，之前你也这么做过。
2. 使用JSON数据初始化一个静态的 `Topapps` 对象
3. 得到feed中的第一项，也就是第一个app
4. 在控制台输出app

讲真，这是你需要的所有代码。

保存并运行storyboar你会看到再次成功获得app的名字，不过这次是以更优雅的方式：

```
App(name: "Game of War - Fire Age", link: "https://itunes.apple.com/us/app/game-of-war
-fire-age/id667728512?mt=8&uo=2")
```

这里关注的是解析本地数据，你如何解析一个远程的数据？

检索远程JSON

是时候将这个工程变得更加实用了。通常会检索远程数据而不是本地的。可以通过网络请求轻易的得到AppStore的排行榜。

打开 `Datamanager.swift` 并定义Top Apps URL：

```
let TopAppURL = "https://itunes.apple.com/us/rss/topgrossingipadapplications/limit=25/
json"
```

接着添加如下方法:

```
public class func getTopAppsDataFromItunesWithSuccess(success: ((iTunesData: NSData!)
-> Void)) {
  //1
  loadDataFromURL(NSURL(string: TopAppURL)!, completion:{(data, error) -> Void in
      //2
      if let data = data {
        //3
        success(iTunesData: data)
      }
  })
}
```

上面的代码看起来很熟悉，不过与检索本地文件不同的是，你使用 `NSURLSession` 从iTunes上获得了数据。里面发生了什么：

1. 当第一次调用 `loadDataFromURL` 时，使用 `NSData` 对象中的的URL与一个completion闭包作为输入参数。
2. 接着使用可选绑定确保data存在
3. 最后将data传递给success

打开storyboard并将下面这行:

```
DataManager.getTopAppsDataFromFileWithSuccess { (data) -> Void in
```

替换为

```
DataManager.getTopAppsDataFromItunesWithSuccess { (data) -> Void in
```

现在你可以从iTunes上检索真实的数据了。

保存并运行你的storyboard，你将会看到数据解析的结果依然是第一个app：

```
App(name: "Game of War - Fire Age", link: "https://itunes.apple.com/us/app/game-of-war
-fire-age/id667728512?mt=8&uo=2")
```

上面的值可以会与你的不同，毕竟AppStore的top apps是会变的。

一般人们不仅会对AppStore的top app感兴趣，他们想看到所有top apps的一个列表。为了满足这个不需要写任何额外的代码，可以使用下面这个代码段轻易的实现：

```
topApps.feed?.entries
```

Gloss的工作原理

可以看到Gloss在解析处理方面的卓越成效，不过它是怎么工作的呢？

<~~是 `Decoder.decode` 功能中的一个 `自定义操作符` (custom operator)。Gloss中支持多种类型的解码：

- Simple types (Decoder.decode)
- Decodable models (Decoder.decodeDecodable)
- Simple arrays (Decoder.decode)
- Arrays of Decodable models (Decoder.decodeDecodableArray)
- Enum types (Decoder.decodeEnum)
- Enum arrays (Decoder.decodeEnumArray)
- NSURL types (Decoder.decodeURL)
- NSURL arrays (Decode.decodeURLArray)

> 如果你想了解关于自定义操作符的更多信息，可以看看这个教程：Operator Overloading in Swift Tutorial

在这个教程中严重依赖 `Decodable` 模型。如果你需要实现更复杂的类型，可以扩展 `Decoder` 并提供你自己的解码implementation

当然Gloss也可以将对象转换为JSON。如果你对这个感兴趣可以看看 `Encodable` 协议。

这里是教程的完整playground

Apple Pencil Tutorial: Getting Started

Apple Pencil指南：入门

> - 原文链接：Apple Pencil Tutorial: Getting Started
> - 原文作者：Caroline Begbie
> - 译者：yrq110

> 更新至Xcode 7.3、iOS9.3与Swift 2.2 04-01-2016

我知道你们大多数人都有一台搭配一只Pencil（触控笔，以下统一用Pencil）的iPad Pro（话外音:并没有）

也许你会像我一样，一旦感受到使用Pencil进行绘画的魅力，就会想将其添加到你的所有app中。

当我买了第一台iPad时我就一直在等待一个类似这样的设备。从我的涂鸦中可以看到虽然我并不是伦勃朗（17世纪荷兰画家）不过我发现用Pencil来做笔记是很棒的。我能想象出那些艺术工作者会使用Apple Pencil创造出各种各样令人惊叹的作品。

在这篇Apple Pencil教程中，你会将会学到Pencil中的细节。这里是你需要学习的一些关键点：

- 如何使用force
- 如何提高准确度
- 如何实现阴影效果
- 如何添加一个橡皮擦
- 如何提升使用预测与实际绘图的体验

到了这篇教程的最后，你就做好了将Apple Pencil集成到你app的准备了！

目录

- 倾斜Pencil
    - 高度、方位与单位向量
- 绘制阴影
- 使用纹理
- 使用方位角调整宽度
- 玩玩透明度
- 降低延迟

前提

学习本教程，你需要：

- 一台iPad Pro与Apple Pencil。在模拟器中无法测试Pencil，而且只能在iPad Pro上使用，老版的iPad上无法使用。听起来你有一个升级设备的理由了！
- 最低Xcode 7.1与iOS 9.1
- 掌握Core Graphics。你需要知道什么是context，如何创建它，怎么进行线条绘图。看看这个the first part of our Core Graphics tutorial ，会使你快速熟悉，那个app还会提醒你喝水:)

入门

在这篇教程中，你将会创建一个叫做Scribble的app。这是一个可以让你使用响应式UI、压力灵敏度与阴影进行绘图的简易app。

下载Scribble。在你的iPad Pro上试试，用Pencil与手指进行绘图。

晃动iPad清屏，就像一个Etch-A-Sketch！

原理就是，Scribble中具有一个能识别手指或Pencil触摸的画板视图，并且根据你触摸的变化实时更新显示的视图。

看看CanvasView.swift中的代码。

最关键的代码在touchesMoved(_:withEvent:)中，当用户与画板视图交互会被触发。这个方法创建了一个CGContext并且将其中的图像绘制在负责显示的画板中。

touchesMoved(*:withEvent:)*中会调用*drawStroke*(:touch:)在context中的前一触摸与当前触摸之间画一条线

touchesMoved(_:withEvent:)将画板视图显示的图像替换为context中最新的图像。

看到没？就是这么简单。

第一次使用Pencil绘制

就算在数字环境中，用你的手指绘画从未是一件优雅的事。使用Pencil来绘制就像传统的模拟方法，使用一个铅笔与纸张的基础UI。

现在准备好见识Pencil的第一个特性——force。当你用力触摸屏幕时得到的线条就比较宽。这个特性并不适用于你的手指，这里有一个小九九之后你就会看到。

Force的大小被记录在touch.force中。一个平均力度触摸的force是1.0，你需要用force乘以某个值来产生正确的线条宽度。更多详情请见下文

打开CanvasView.swift在类的最前面添加如下常量:

```
private let forceSensitivity: CGFloat = 4.0
```

你可以调整这个forceSensitivity量，使线条宽度的变化对压力更敏感。

找到lineWidthForDrawing(_:touch:)，这个方法用来计算线条的宽度。

在return语句前添加如下代码:

```
if touch.force > 0 {
  lineWidth = touch.force * forceSensitivity
}
```

这里你通过计算触摸点的force与forceSensitivity的乘积来得到线条宽度，不过记住这仅适用于Pencil。如果你使用手指的话，touch.force将会是0，你就不能改变线条宽度了。

运行一下。使用Pencil画一些线条，注意线条是怎样随着你触摸屏幕的力度变化而变化的。

更加平滑的绘制

你会注意到当你绘制时，线条中会有一些尖锐的过渡点而不像自然的曲线那样。在以前，你也许会做一些像将线条转换为样条曲线这样的复杂工作才能得到适宜的图案，不过Pencil及其简化了这些工作。

Apple告诉我们iPad Pro扫描触摸事件的速率是120次/秒，将Pencil在屏幕附近时扫描速率会翻倍达到240次/秒。

iPad Pro的刷新速率是60Hz。这意味着理论上识别出两个触摸点时将只能显示一个。另外，如果在当前场景下有大量事务需要处理时，一些帧的触摸事件会全部丢失因为被占用的主线程无法处理它。

尝试快速的画一个圆。它应该是圆的，不过结果看起来像有一堆锯齿点的多边形:

为了解决这个问题，Apple提出了一个合并触摸(coalesced touches)的概念。捕捉那些丢失的触摸放进一个新的UIEvent数组中，你可以通过coalescedTouchesForTouch(_:)来访问。

找到CanvasView.swift中的touchesMoved(_:withEvent:)，将:

```
drawStroke(context, touch: touch)
```

替换为:

```
// 1
var touches = [UITouch]()

// 2
if let coalescedTouches = event?.coalescedTouchesForTouch(touch) {
  touches = coalescedTouches
} else {
  touches.append(touch)
}

// 3
print(touches.count)

// 4
for touch in touches {
  drawStroke(context, touch: touch)
}
```

让我们来逐段分析:

1. 首先，你创建了一个新数组用来放置你需要处理的所有触摸点。
2. 检查合并触摸，如果存在则将他们存放进新的数组
3. 添加一个log来看看你需要处理的触摸点数量
4. 最后，对于新数组中每一个触摸点都要调用一次drawStroke(_:touch:)

运行一下。用你的Pencil画一些优美的花饰，沉醉于黄油般的光滑与笔触宽度调节中:

将你的注意转到调试窗口。你将会发现当你使用Pencil绘制而不是手指时，你会得到更多的触摸点。

你还会发现使用Pencil画的圆更圆，这就是由于iPad Pro感应到Pencil时会扫描出双倍的触摸点。

倾斜Pencil

现在你可以熟练的在app里作画了。然而如果你读过任何Pencil评测的话，你会记得讨论过它有阴影绘制的能力。用户们需要做的就是将笔倾斜，殊不知这种阴影是不会自动产生，取决于我们这些聪明的开发者如何利用代码去让它工作。

高度、方位与单位向量

在这节中，我会向你描述如何测量这种倾斜，你会在下一节添加简易的阴影效果。

当你使用Pencil时，你可以在三维空间中旋转它。上下的方向被叫做高度(altitude),两侧被叫做方位(azimuth):

高度角属性是UITouch在iOS9.1中新添加的属性，仅适用于Apple Pencil。它的单位是弧度。当Pencil平放在iPad表面时高度角就是0.当它垂直于屏幕时高度角是π/2。记住360度对应的是2π弧度，因此π/2就是90度。

UITouch有两个新方法用来获取方位角:

azimuthAngleInView(*:)与azimuthUnitVectorInView(*:)。根据你的情况选择相应的方法。

感受一下方位角单位向量是如何工作的。仅供参考，一个单位向量的长度为1，从坐标(0,0)点指向一个方向。

在touchesMoved(_:withEvent:)的最上面，在guard语句之后添加:

```
print(touch.azimuthUnitVectorInView(self))
```

运行一下。

在调试控制台中很难得到满足精度要求的值，不过向量大约在x方向为1，在y方向为0，即(1,0)。

将Pencil逆时针旋转90度，笔尖朝着iPad的下方，此时的方向大约是(0,-1)。

注意x方向使用cos计算，y方向使用sin。比如说你像上图那样握笔，从你初始的水平方向逆时针旋转45度，单位向量为(cos(45),sin(-45))或(0.7071,-0.7071)。

> 如果你不太了解向量，这是一个需要补充的知识。这个教程会使你充分了解向量。

当你理解Pencil方向的改变与指示向量的关系后删掉最后的print语句。

绘制阴影

现在你知道了如何测量倾斜量，已经准备好在Scribble中添加简单的阴影了。

当Pencil处于一个自然绘图的角度，使用力道来决定笔画的粗细，当用户倾斜到一边时，可以用力道来测量阴影的透明度。

你可以根据线条的方向与握笔的方向来计算线条的粗细。

如果你还不太明白试着找一支铅笔与一张纸，将铅笔倾斜到一边进行绘制，使笔尖与纸张的接触达到最大。当你沿着铅笔倾斜的方向绘制时，阴影是较细的。当你垂直于倾斜方向进行绘制时，阴影是最粗的。

使用纹理

首先要做的是改变线条的纹理使其看着像真正铅笔的阴影。为了达到这个效果我们使用开始工程资源中的PencilTexture图片。

在CanvaView中添加这个属性：

```
private var pencilTexture = UIColor(patternImage: UIImage(named: "PencilTexture")!)
```

这里允许你使用pecilTexture作为颜色来绘制，而不是之前默认的红色。

在drawStroke(_:touch:)中找到如下这行:

```
drawColor.setStroke()
```

改为:

```
pencilTexture.setStroke()
```

运行一下。嘿！你的线条更像一个铅笔的线条了:

> 在这篇教程中，你使用了一种很简单的方式来实现纹理。在功能齐全的艺术类app中的笔刷引擎是很复杂的，不过入门的话这样做足够了。

检测一下Pencil倾斜的程度已经足够产生阴影，将这个常量添加进CanvasView中:

```
private let tiltThreshold = π/6  // 30º
```

如果你发现这个值没有起作用，你需要自己设置一个合适的值。

> π是一个方便的常量，在CanvasView.swift中定义为CGFloat(M_PI)
>
> 在图形编程中，使用弧度是很重要的而不是通过角度为中介进行转换。看看wiki的这个图片，了解一下角度与弧度间的关系。

接下来，在drawStroke(_:touch:)方法中找到这一行:

```
let lineWidth = lineWidthForDrawing(context, touch: touch)
```

改为:

```
var lineWidth:CGFloat

if touch.altitudeAngle < tiltThreshold {
  lineWidth = lineWidthForShading(context, touch: touch)
} else {
  lineWidth = lineWidthForDrawing(context, touch: touch)
}
```

这里你添加了一个if用来检测Pencil倾斜的角度是否超过π/6或30度。如果未超过的话就会调用阴影绘制方法而不是线条绘制方法。

现在将这个方法添加到CanvasView的底部:

```
private func lineWidthForShading(context: CGContext?, touch: UITouch) -> CGFloat {

  // 1
  let previousLocation = touch.previousLocationInView(self)
  let location = touch.locationInView(self)

  // 2 - vector1 is the pencil direction
  let vector1 = touch.azimuthUnitVectorInView(self)

  // 3 - vector2 is the stroke direction
  let vector2 = CGPoint(x: location.x - previousLocation.x, y: location.y - previousLo
cation.y)

  // 4 - Angle difference between the two vectors
  var angle = abs(atan2(vector2.y, vector2.x) - atan2(vector1.dy, vector1.dx))

  // 5
  if angle > π {
    angle = 2 * π - angle
  }
  if angle > π / 2 {
    angle = π - angle
  }

  // 6
  let minAngle: CGFloat = 0
  let maxAngle = π / 2
  let normalizedAngle = (angle - minAngle) / (maxAngle - minAngle)

  // 7
  let maxLineWidth: CGFloat = 60
  var lineWidth = maxLineWidth * normalizedAngle

  return lineWidth
}
```

这儿的数学有点复杂，让我们来详细的分析一下:

1. 保存前一触摸点与当前触摸点
2. 保存Pencil的方位向量
3. 保存当前笔触的方向向量
4. 计算画线与Pencil方向的角度差
5. 限制角度在[0,90]的范围内。如果是90度则线条最宽。记住所有计算都是使用弧度的，π/2即为90度
6. 归一化角度，
7. 计算线宽最大值与归一化角度的乘积得到正确的阴影宽度

> 注意 不论何时使用Pencil，运用下面的公式都是很方便的:
>
> 向量角:angle = atan2(opposite, adjacent)
>
> 归一化:normal = (value - minValue) / (maxValue - minValue)

运行一下。像图中标示的那样握笔，不改变角度，画些阴影出来。

注意笔触的方向是如何改变线宽的。这些做虽然有点幼稚，不过你肯定会发现其中的潜力的。

使用方位角调整宽度

还有一件事要做:当你使用铅笔以90度的角度进行绘制时，线宽会比其它倾斜的角度时都窄。不过当你使用Apple Pencil时宽度则不会改变。

在CanvaView类的顶部添加这个常量:

```
private let minLineWidth: CGFloat = 5
```

它定义了一个阴影线条的最窄宽度，你可以根据自己的喜好调整这个数值。

在lineWidthForShading(_:touch:)方法的底部，在return语句前添加以下语句:

```
// 1
let minAltitudeAngle: CGFloat = 0.25
let maxAltitudeAngle = tiltThreshold

// 2
let altitudeAngle = touch.altitudeAngle < minAltitudeAngle ? minAltitudeAngle : touch.
altitudeAngle

// 3
let normalizedAltitude = 1 - ((altitudeAngle - minAltitudeAngle) / (maxAltitudeAngle -
 minAltitudeAngle))

// 4
lineWidth = lineWidth * normalizedAltitude + minLineWidth
```

> 注意：确保你将这段代码添加进的是lineWidthForShading(*:touch:)*方法，而不是*lineWidthForDrawing*(:touch:)

这里有大量需要消化的东西，让我们捋一捋:

1. 理论上，Pencil最小的高度是0度，意味着Pencil会平躺在iPad上，笔尖没有接触屏幕，

这样的话高度就不会记录下来。实际情况下的最小高度大约为0.2，这里我设置为0.25

2. 如果高度小于最小值，则使用最小值
3. 像之前做的那样，将高度值归一化在[0,1]的范围内
4. 最后，计算使用方位角算出的线宽与归一化角度的乘积，再加上最小线宽

运行一下。在绘制阴影时改变Pencil的高度观察线条的粗细是如何变化的。逐渐使Pencil的高度增加会使你的线条变得更加光滑。

玩玩透明度

本节中的最后一个任务是根据力道来改变纹理的透明度，使阴影看起来更加真实。

在lineWidthForShading(_:touch:)方法中的return语句前添加以下代码:

```
let minForce: CGFloat = 0.0
let maxForce: CGFloat = 5

let normalizedAlpha = (touch.force - minForce) / (maxForce - minForce)

CGContextSetAlpha(context, normalizedAlpha)
```

经过之前的代码，这个的作用就显而易见了。使用了force属性并且将其归一化在[0,1]范围内，接着使用这个值设置context的alpha值。

运行一下。用不同的压力来绘制:

你可能会像我一样素描时在某处会有一些失误，并希望可以擦除这些错误的线条。

在这一节中，你将会看到如何区分Apple Pencil与手指的使用。更进一步说，你可以将app设置成使用手指扮演一个忠实的橡皮擦。

检测使用的是Pencil还是手指很简单，使用UITouch中的type属性即可。

在CanvasView的顶部，添加一个橡皮擦颜色的属性。你将会在画布视图上使用背景色进行绘制，这会出现一个橡皮擦的效果。是不是很聪明?:

```
private var eraserColor: UIColor {
  return backgroundColor ?? UIColor.whiteColor()
}
```

这里你将橡皮擦颜色设为了视图的背景色，如果它非空的话。

接下来，找到drawStroke(_:touch:)方法中的如下代码:

```
if touch.altitudeAngle < tiltThreshold {
  lineWidth = lineWidthForShading(context, touch: touch)
} else {
  lineWidth = lineWidthForDrawing(context, touch: touch)
}

pencilTexture.setStroke()
```

使用如下代码替换:

```
if touch.type == .Stylus {
  if touch.altitudeAngle < tiltThreshold {
    lineWidth = lineWidthForShading(context, touch: touch)
  } else {
    lineWidth = lineWidthForDrawing(context, touch: touch)
  }
  pencilTexture.setStroke()
} else {
  lineWidth = 20
  eraserColor.setStroke()
}
```

这里添加了一个if用来检测是Pencil还是手指，决定之后是改变线宽还是使用橡皮擦颜色绘制。

运行一下。现在你可以用你的手指来清理那些不整齐的边界或任何其他东西。

在iOS8之后，UITouch中加入了一个叫做majorRadius的属性，如它的名字所示，保存触摸时的尺寸。

找到前面你添加的这段代码:

```
lineWidth = 20
```

替换为这个:

```
lineWidth = touch.majorRadius / 2
```

运行一下。绘制一片阴影区域，然后用手指的指尖与指面进行擦拭，观察粗细的变化。

当你使用优雅的Apple Pencil画图后会发现用手指绘制是多么的笨拙与痛苦。

降低延迟

你也许会认为线条的绘制会紧随着笔尖的滑动。并不完全是那样，在触摸与线条渲染之间是有延时的。对此Apple使用了一个技巧:触摸预测

看起来很难以置信，Apple知道你的Pencil、手指在哪，将要画什么。那些预测点会保存进一个UIEvent的数组中，你可以预先绘制好预测点。是不是很酷？

在你使用预测点之前有一个小小的技术障碍需要克服。当前我们在图形上下文画的线会立即在画布视图中显示。

你需要在画布中绘制这些预测点不过当真实点被捕捉到时需要丢弃那些预测点。

举个例子，当你根据预测的曲线画一个S型图案，不过当你改变方向时，这些预测会出错并且需要被舍弃。这张图片说明了问题，绘制的S是红色的，触摸的预测显示为蓝色。

你的代码需要做如下工作来避免这个问题:

1. 你需要创建一个名为drawingImage的UIImage属性来从图形上下文中捕捉那些真实的触摸，而不是预测的
2. 每一个触摸的移动事件都会在图形上下文中绘制drawingImage
3. 真实的触摸会绘制在图形上下文中，将会保存进drawingImage中而不是画布视图的image属性
4. 预测触摸会被绘制进图形上下文中
5. 在图形上下文绘制完预测触摸后，赋值给canvasView.image-给用户展现的视图

这样就不会有预测点被绘制进drawingImage中，而且每次有触摸移动事件发生时，预测点都会被删除。

Introduction to Functional Programming in Swift

Swift函数式编程介绍

> - 原文链接：Introduction to Functional Programming in Swift
> - 原文作者：Joe Howard
> - 译者：yrq110

在2014年WWDC上隆重登场的swift不仅仅是作为一门新的语言，它为iOS与OS X平台带来了许多新的特性。

这篇教程会针对其中的FP(Functional Programming)，介绍函数式的思想与技术。

> 注意。如果你想深入了解swift的函数式编程，可以看看Swift by Tutorials中的函数式编程章节，这里也可以。

目录

- 函数式方法2

入门

在Xcode中创建一个新的playground命名为IntroFunctionalProgramming，其他选项默认，选择iOS平台，点击create创建。

记住，这篇教程所讲的函数式编程是高度抽象的， 因此将这些概念想象成现实中的场景会很有帮助。在这里，想象一下你在为一个游乐园做一个app，游乐园的游乐设施数据由远程服务器的API提供。

使用如下代码替换playground中的模板代码:

```
enum RideType {
  case Family
  case Kids
  case Thrill
  case Scary
  case Relaxing
  case Water
}

struct Ride {
  let name: String
  let types: Set<RideType>
  let waitTime: Double
}
```

这里你创建了一个叫RideType的枚举，包含了一些游乐设施的类型。还有一个叫Ride的结构体，包含设施的名称，类型的集合与等待时间。

现在你也许会想象出玩海盗船时失重的恐怖，过山车的激情。不过相信我，没有一种设施的感觉像从其他编程风格到函数式编程的转换。

什么是函数式编程？

函数式编程与其他表现方式的区别常常被描述为命令式与声明式思想的区别。

- 命令式思想在解决问题时思考的是算法与步骤，或者说如何从A(方法)到B(解决方案)。一个典型的命令式的例子：一个纸杯蛋糕的食谱，告诉你放多少材料，怎么去混合，怎么去烘焙。
- 声明式思想分析一个问题考虑解决方案是什么，而不是如何得到解决方案。用一个食谱的例子来说明这个概念。一个声明式的食谱会给你一张纸杯蛋糕的图片与一段描述，而不是告诉你如何混合成分与烘焙。

函数式编程鼓励使用基于函数的声明方法来解决问题。在这篇教程中你将会看到更多的命令式与声明式编程的区别。

函数式编程概念

在这一节中，你将会学习函数式编程中的重要概念。许多函数式编程的专著中都挑选了不变状态(immutable state)与无副作用(lack of side effects)作为最重要的方面，为何我们不从这里开始呢？

不变性与副作用

不管你最开始学的什么编程语言，接触到的第一个概念应该就是用变量来表现数据。当你回到这种表现的最初理念时，变量就会变得很古怪。

在你的playground中添加如下看起来理所当然的代码:

```
var x = 3
// other stuff...
x = 4
```

作为一个开发者在学习面向过程或面向对象时就是这么讲的，你不会花一点时间会思考它。不过一个量如何能先等于3然后等于4？

变量意为一个变化的量。考虑数学中的量x，你会在软件行为中将时间作为关键参数。你在改变变量时创造了可变的状态。

若在一个相对简单的系统中，可变状态也许不是一个大问题。不过当连接多个对象时，像一个庞大的OOP系统，可变状态就会很让人头痛。

例如，当2个或多个线程同时访问一个变量时，它们也许在访问或修改时出错，出现不期望的行为。像竞态条件、锁死等问题。

想象一下你可以在一个状态从不改变的地方写代码。在并发系统中的大量问题都会消失——poof！你可以创建一个不变属性来实现它，或者数据在程序运行期间不允许更改。

在objective-C中可以通过一个常量来实现，不会你的属性在创建时默认是可变模式。在swift中默认是不变的。

使用不变数据的优点就是在代码单元中使用时将会没有副作用，意味着你代码中函数不会改变它们外部的元素，当调用函数时不会发生可怕的现象。

在你playground的Ride结构体后面添加如下数组，在这篇教程中使用一个不变的swift常量来表现你的主要数据，看看会发生什么:

```
let parkRides = [
  Ride(name: "Raging Rapids", types: [.Family, .Thrill, .Water], waitTime: 45.0),
  Ride(name: "Crazy Funhouse", types: [.Family], waitTime: 10.0),
  Ride(name: "Spinning Tea Cups", types: [.Kids], waitTime: 15.0),
  Ride(name: "Spooky Hollow", types: [.Scary], waitTime: 30.0),
  Ride(name: "Thunder Coaster", types: [.Family, .Thrill], waitTime: 60.0),
  Ride(name: "Grand Carousel", types: [.Family, .Kids], waitTime: 15.0),
  Ride(name: "Bumper Boats", types: [.Family, .Water], waitTime: 25.0),
  Ride(name: "Mountain Railroad", types: [.Family, .Relaxing], waitTime: 0.0)
]
```

当你使用let创建parkRides而不是用var时，这个数组及它的项就是不可变的。使用如下方法尝试改变数组中的一项:

```
parkRides[0] = Ride(name: "Functional Programming", types: [.Thrill], waitTime: 5.0)
```

改变其中的项会发生编译错误。去修改那些ride吧！没门儿，小鬼！

模块化

你将会写出你的第一个函数，在swift的String使用NSString的方法，因此需要在playground的头部添加Foundation框架:

```
import Foundation
```

假设你需要一个按字母表排序的设施名称列表，使用命令式的方法来实现的话，添加如下函数到你的playground底部:

```
func sortedNames(rides: [Ride]) -> [String] {
  var sortedRides = rides
  var i, j : Int
  var key: Ride

  // 1
  for (i = 0; i < sortedRides.count; i++) {
    key = sortedRides[i]

    // 2
    for (j = i; j > -1; j--) {
      if key.name.localizedCompare(sortedRides[j].name) == .OrderedAscending {
        sortedRides.removeAtIndex(j + 1)
        sortedRides.insert(key, atIndex: j)
      }
    }
  }

  // 3
  var sortedNames = [String]()
  for ride in sortedRides {
    sortedNames.append(ride.name)
  }

  print(sortedRides)

  return sortedNames
}
```

这里你做了:

1. 在函数内遍历设施
2. 执行插入排序
3. 取得排序后设施中的名称

从sortedNames(:)的角度来看，它接收了设施的列表，然后输出了排序后的包含设施名称的列表。sortedNames(:)外面的量没有受到影响，为了证明，首先我们打印出排序后的设施名称:

```
print(sortedNames(parkRides))
```

然后收集设施列表中的名称并打印出来:

```
var originalNames = [String]()
for ride in parkRides {
  originalNames.append(ride.name)
}

print(originalNames)
```

在编辑器中你会发现sortedNames(:)中对设施的排序并没有改变输入的设施列表。你使用命令式代码创建的这个模块函数被认为是准函数式(quasi-functional)的。

即使如此，用命令式语句写出的函数依旧很冗长、笨拙。如果有一种技术，可以通过使用像sortedNames(:)这样的函数更进一步的简化代码，岂不是很棒？

第一类与高阶函数

另一个函数式编程的特点就是函数们。第一类函数是其中一种，它意味着可以被分配给一个值，可以作为其它函数的输入与输出。再往上就是高阶函数，它可以将函数作为参数并返回其它函数。

在这一节中，你将会学习到函数式编程中几个常见的高阶函数——filter、map与reduce。

Filter

在swift中，filter(:)是CollectionType型值的一个方法，比如说数组(array),它可以将另一个函数作为参数输入，若数组中的值执行这个函数后的值为眞，则返回。

filter(:) 将输入的函数应用到数组中的每一个元素上，返回另一个包含返回值为眞的所有元素的数组。

回想一下你的sortedNames做过什么，用声明式的思想去思考一下而不是命令式。

注释掉sortedNames，你将会更有效率的重写它。在playground底部创建一个将Ride作为参数的函数:

```
func waitTimeIsShort(ride: Ride) -> Bool {
  return ride.waitTime < 15.0
}
```

waitTimeIsShort(:)接受一个ride，如果等待时间小于15秒则返回true，否则返回false。

在parkRides上调用filter然后输入你刚刚创建的函数:

```
var shortWaitTimeRides = parkRides.filter(waitTimeIsShort)
print(shortWaitTimeRides)
```

你把shortWaitTimeRides定义为一个变量因为你会在playground中重用它，会在其它地方方访问。在playground的输出中，你会看到filter输出中的Crazy Funhouse与Mountain Railroad是正确的结果。

在filter的尾部使用一个闭包(closure)函数可以得到同样的结果:

```
shortWaitTimeRides = parkRides.filter { $0.waitTime < 15.0 }
print(shortWaitTimeRides)
```

在这里，.filter()会去除parkRides数组中的每一个ride，观察它的waitTime属性并计算它是否小于15。你声明式地告诉程序你想让它做什么而不是怎么做，这一开始看起来有点神秘。

Map

CollectionType类型的方法map(:)也接受一个函数作为参数，在对每一个参数中的元素进行操作后返回一个与之前相同长度的数组。map函数的返回值类型可以不与参数元素的类型相同。

对parkRides数组中的元素执行map，得到一个包含所有设施名称字符串的列表:

```
let rideNames = parkRides.map { $0.name }
print(rideNames)
```

你可以像下面那样排序并打印出名称列表，使用sort来进行MutableCollectionType类型的排序:

```
print(rideNames.sort(<))
```

你的sortedNames(:)方法得益于map、sort与一个闭包，现在变成了仅仅两行！

Reduce

CollectionType类型方法reduce(:,:)接受两个参数。第一个是一个类型T的值，第二个是函数，它将T的值与另一个元素融合产生一个具有新值的T，输入函数会处理调用的集合中的每一个元素。

假设你想知道公园里所有设施的平均等待时间。

在reduce中输入值0.0，然后使用一个闭包函数对每一种设施等待时间与设施总数的比值求和。

```
let averageWaitTime = parkRides.reduce(0.0) { (average, ride) in average + (ride.waitTime/Double(parkRides.count)) }
print(averageWaitTime)
```

在这个例子中，最初的两次迭代如下所示:

| iteration | average | ride.waitTime / Double(parkRides.count) | resulting average ||—|—|—|—||1|0.0|45.0 / 8.0 = 5.625 | 0.0 + 5.625 = 5.625||2|5.625|10.0 / 8.0 = 1.25 | 5.625 + 1.25 = 6.875|

如你所见，平均值的结果使用不断迭代的平均值。这个过程会持续到遍历完parkRides中的每一个设施才会结束。它使你只用了一行代码就能得到平均值！

柯里化

> 注意。随着现在swift的开源，在swift3.0中貌似会移除柯里化函数语句。你可以在这里找到信息，尤其是**Detailed design**部分

函数式编程中又一个高级特性就是柯里化，由数学家Haskell Curry得名。柯里化的思想是将接受多个参数的函数作为一个接受单个参数的函数应用的序列。

明确一下，在这篇教程中，柯里化并不是一个好吃到流泪的咖喱，不过你可以通过创建使用括号将两个参数分开的双参数函数来实现一下柯里化:

```
func rideTypeFilter(type: RideType)(fromRides rides: [Ride]) -> [Ride] {
  return rides.filter { $0.types.contains(type) }
}
```

这里，rideTypeFilter(:)(:)接受了一个设施类型作为第一个参数，一个设施的数组作为第二个参数。

构造另一个函数是柯里化的一个较好应用。使用单个参数来调用你的柯里化函数，然后构造一个单个参数的函数。

在你的playground中添加这个设施类型的过滤器，动手试试:

```
func createRideTypeFilter(type: RideType) -> [Ride] -> [Ride] {
  return rideTypeFilter(type)
}
```

注意到createRideTypeFilter返回映射一个ride数组到另一个ride数组的函数，通过调用一个单个参数的rideTypeFilter。使用createRideTypeFilter 为孩子们创建一个过滤器，然后使用过滤器过滤你的公园设施表，像这样:

```
let kidRideFilter = createRideTypeFilter(.Kids)

print(kidRideFilter(parkRides))
```

在playground的输出中，你会看到kidRideFilter过滤掉了所有非孩童类的设施。

纯函数

纯函数的思想，是函数式编程中一个重要的概念。

一个纯函数应该符合以下两个标准:

- 对于相同的输入总会产生相同的输出结果，输出仅仅取决于输入
- 在函数外没有任何副作用

纯函数的存在于不变状态的使用密切相关。

在你的playground中添加如下的纯函数:

```
func ridesWithWaitTimeUnder(waitTime: Double, fromRides rides: [Ride]) -> [Ride] {
  return rides.filter { $0.waitTime < waitTime }
}
```

ridesWithWaitTimeUnder(:fromRides:)是一个纯函数，因为输入一个相同的等待时间阈值与设施列表后总会得到相同的输出。

使用纯函数可以方便地对一个函数进行单元测试，在你的playground中加入如下的断言语句来模拟一个单元测试:

```
var shortWaitRides = ridesWithWaitTimeUnder(15.0, fromRides: parkRides)
assert(shortWaitRides.count == 2, "Count of short wait rides should be 2")
print(shortWaitRides)
```

在这里测试了当输入15.0的等待时间阈值与parkRides时，是否总会输出2个设施。

引用透明

引用透明的概念与纯函数密不可分。对于一个x，无论你怎么调用f(x)总会得到一个相同的结果(不受外部因素与变量影响)，则称f是引用透明的。它用来预测代码并允许编译器去执行优化行为。

纯函数满足这个条件。如果你很熟悉Objective-C的话会发现#define与macros就是引用透明的例子。

用纯函数ridesWithWaitTimeUnder的函数体去替换它看看它是不是引用透明的:

```
shortWaitRides = parkRides.filter { $0.waitTime < 15.0 }
assert(shortWaitRides.count == 2, "Count of short wait rides should be 2")
print(shortWaitRides)
```

递归

最后一个要讨论的概念就是递归，一个调用自己函数体的函数行为。在函数式语言的声明式语句中，使用递归可以有效地替代循环结构。

当一个函数的输入时再次调用本身时，就是递归的情况。为了防止函数的无限调用，递归函数需要一个边界条件去终止过程。

使用满足Comparable协议的Ride结构体扩展来为你的设施添加一个递归排序的函数，

```
extension Ride: Comparable { }

func <(lhs: Ride, rhs: Ride) -> Bool {
  return lhs.waitTime < rhs.waitTime
}

func ==(lhs: Ride, rhs: Ride) -> Bool {
  return lhs.name == rhs.name
}
```

按Swift操作符重载的要求，你需要在extension外添加Comparable操作符。当等待时间更短时认为一个设施小于另一个，当设施名字相同时认为两者相同。

现在，在你的playground中添加一个递归快排的方法:

```
func quicksort<T: Comparable>(var elements: [T]) -> [T] {
  if elements.count > 1 {
    let pivot = elements.removeAtIndex(0)
    return quicksort(elements.filter { $0 <= pivot }) + [pivot] + quicksort(elements.f
ilter { $0 > pivot })
  }
  return elements
}
```

quicksort(:)是一个接受Comparable数组并根据一个基准值进行排序的泛型函数。

添加如下代码检查设施列表的排序情况:

```
print(quicksort(parkRides))
print(parkRides)
```

第二行证明了quicksort(:)并没有修改输入的不变数组。

命令式 vs 声明式

考虑一下下面的问题，结合你学习到的函数式编程知识，对命令式与函数式编程的不同之处会有一个清晰的理解。

> 问题陈述 一个带小孩的家庭想在频繁如厕的情况下玩尽可能多的设施，所以他们需要找到排队最短的，适合小孩的设施。帮助他们找到所有等待时间小于20分钟的家庭类设施并以等待时间为基准排序(升序)。

命令式方法

暂时忘掉所有所学的函数式编程，想想通过怎样的算法来解决这个问题。你也许会:

1. 创建一个设施的空数组
2. 遍历所有设施找到包含属性Family的项
3. 当一个家庭类设施的等待时间小于20分钟，则加入到可变数组中
4. 最后用等待时间对设施进行排序

这就是解决问题的答案步骤，以下是算法的实现，请添加到你的playground中:

```
var ridesOfInterest = [Ride]()
for ride in parkRides {
  var typeMatch = false
  for type in ride.types {
    if type == .Family {
      typeMatch = true
      break
    }
  }
  if typeMatch && ride.waitTime < 20.0 {
    ridesOfInterest.append(ride)
  }
}

var sortedRidesOfInterest = quicksort(ridesOfInterest)

print(sortedRidesOfInterest)
```

会看到Mountain Railroad, Crazy Funhouse与Grand Carousel是最好的选择，并且它们按照等待时间递增的顺序排好了序，使用quicksort(:)函数去排序。

用命令式代码书写可以是可以，但是扫一眼后对它所做的事情并没有一个清晰的思路。需要暂停窥视算法细节去想想这个。你说没什么大不了的?如果你回过头来维护，debug或者将它交给其他开发者呢？就像网上那些人说的：你这样做不对！

函数式方法1

函数式编程可以做得更好，在你的playground中添加如下代码:

```
sortedRidesOfInterest = parkRides.filter({ $0.types.contains(.Family) && $0.waitTime <
20.0 }).sort(<)
```

使用包含集合的filter方法去寻找匹配的设施，并像以前那样进行了排序。

添加print来证明这一行简单的代码会输出与命令式语句相同的结果:

```
print(sortedRidesOfInterest)
```

你做到了！在一行中，你告诉了swift去算什么，你想要过滤出包含Family属性并且等待时间小于20分钟的有了设施，并且将它们排序。就这样准确并优雅地解决了问题。

函数式编程不仅让你的代码变得更简洁，也让其含义变得更显而易见。我敢打赌你找不到副作用！

函数式方法2

上面的单行代码解决方法也许有些隐晦，不过可以使用一个可修改的变量，利用之前用过的柯里化过滤方法来创建家庭设施过滤器:

```
let familyRideFilter = createRideTypeFilter(.Family)
sortedRidesOfInterest = ridesWithWaitTimeUnder(20.0, fromRides: familyRideFilter(parkR
ides)).sort(<)

print(sortedRidesOfInterest)
```

使用更加友好的方式得到了相同的结果。一旦有了家庭设施过滤器，就可以再次通过一行代码来解决问题。

Sprite Kit Swift 2 Tutorial for Beginners

Sprite Kit初学指南

> - 原文链接 : [Sprite Kit Swift 2 Tutorial for Beginners]
> - 原文作者 : [Ray Wenderlich]
> - 译者 : [yrq110]

> 注意:本文使用的是swift2，并非最新版本。

就像蝙蝠侠与罗宾、超人与路易斯霍恩一样，Sprite Kit与Swift是一对惊艳的组合:

- **Sprite Kit**是一种非常棒的写iOS游戏的方法，它易学、功能强大，并且由苹果全面支持。
- **Swift**是一门易上手的语言，尤其是若你是一位iOS初级开发者的话。

在这篇教程中，你将学会如何用苹果的2D游戏框架-Sprite Kit去写一款简单的2D游戏，使用Swift语言！

你可以按顺序看也可以直接跳到最后的演示项目，那里会出现一些忍者。

> 注意:这篇教程在我心中有特殊的地位，它的初版是我在这个站上发布的第一篇教程，使用Cocos2D与Objective-C所编写的游戏。哎呀嘛呀，时过境迁啊！

目录

- 完成触摸
- 哥们！游戏结束！

Sprite Kit VS Unity

除了Sprite Kit外最火的游戏框架当属Unity。Unity最初是3D引擎，不过现在也全面支持2D了。

所以在你开始之前，我推荐你考虑一下Sprite Kit与Unity，看看哪个更适合你的游戏。

Sprite Kit的优点:

- **iOS**内置 不需要下载任何的库或者外部依赖。 你可以在其中直接使用其他iOS API，像iAd、In-App Purchases等等。without having to rely on extra plugins.
- 可利用你所掌握的技能 如果你懂iOS与Swift开发的话，就可以非常快的上手。
- 苹果出品 这会让你放下心来，因为它将被所有的苹果新产品所支持。比如说你可以不需要挂载地使用相同的Sprite Kit代码在iOS，OS X与tvOS上运行。
- 免费 可能这是对独立开发者来说最好的原因！ 无需任何费用就可以使用Sprite Kit得所有功能。Unity虽然有免费版本不过并没有专业版本的所有特性(比如说如果你想修改有Unity标志的启动界面的话就需要升级版本)。

Unity的优点:

- 跨平台 这是一个重点。如果你使用Sprite Kit你将会被局限在苹果生态中，使用Unity可以轻松的在安卓、Windows和更多的平台上部署。
- 虚拟场景设计 在Unity中场景的布局与编辑视图层是非常简单的事情，而且可以通过点击一个按钮就能进行游戏的实时测试。Sprite Kit有一个基础的场景编辑器，跟Unity提供的比起来真的是很“基础”。
- 资源包商店 Unity有一个内置的资源包商店，你可以在那里为你的游戏购买各种各样的组件。这些组件有时可以省掉大半的开发时间！
- 更强大 一般来说，Unity拥有比Sprite Kit / Scene Kit更多的特性与功能。

我该选哪个?

看了这么多你可能就想，“好吧，俺该选哪个2D框架？”

答案取决于你的目标是什么。这里是我的两个选择:

- 如果你完全是个新手，或者仅关注苹果生态：使用Sprite Kit - 内置、好学、能找到工作（PS：真的？）。

- 如果你想跨平台，或者做一个复杂的游戏：使用Unity – 更强大、灵活。 If you think Unity is for you, check out some of our Unity written tutorials or our new Unity video tutorial series. 如果你觉得Unity适合你，找一些Unity的教程或者教学视频。

若不是，就开始学习Sprite Kit吧!

Hello, Sprite Kit!

从一个简单的Hello World项目开始，使用Xcode7中的Sprite Kit游戏模板来运行。

打开Xcode，选择File\New\Project，选择iOS\Application\Game模板，然后点击Next:

项目名起为SpriteKitSimpleGame, 语言选择Swift, Game Technology选SpriteKit, 设备是iPhone, 然后点Next:

选择存放项目的地点，点击Create. 选择iPhone 6s模拟器, 点击play按钮运行, 在一个简易的启动界面后你应该会看到下图:

Sprite Kit由场景的概念所组成，就像一个游戏中显示的层级与视图。比如说你可能有一个主要游戏区域的场景与一个拥有层级的世界地图场景。场景与一个拥有层级的世界地图场景。如果你看一眼项目会发现模板已经为你创建好了一个默认的场景 - GameScene。打开GameScene.swift你会看到包含了一段在屏幕中显示文本框的代码，还有一个当你点击就会添加的旋转宇宙飞船。

在这篇教程中主要在GameScene上进行工作。在开始前你需要进行一些调整使你的游戏可以横屏运行而不是竖屏。

初始设置

这个模板对于你来说有两个问题。第一，游戏需要被设置为横屏而它是竖屏的。第二，它正在使用Sprite Kit的场景编辑器，你在这篇教程中并不需要。来让我们解决这些问题。

第一，在工程导航栏中点击SpriteKitSimpleGame，打开target setting，选择SpriteKitSimpleGame target。然后在Deployment Info部分，去掉Portrait使仅有Landscape Left and Landscape Right被选中，像下面这样:

第二，删除GameScene.sks选择送到废纸篓。这个文件允许你布局精灵与可视场景的其他组件，不过对于我们这个游戏可以通过代码轻松创建所以不需要这个。

然后，打开GameViewController.swift用如下代码替换内容:

```
import UIKit
import SpriteKit

class GameViewController: UIViewController {

  override func viewDidLoad() {
    super.viewDidLoad()
    let scene = GameScene(size: view.bounds.size)
    let skView = view as! SKView
    skView.showsFPS = true
    skView.showsNodeCount = true
    skView.ignoresSiblingOrder = true
    scene.scaleMode = .ResizeFill
    skView.presentScene(scene)
  }

  override func prefersStatusBarHidden() -> Bool {
    return true
  }
}
```

GameViewController是一个UIViewController类，它的一个根视图是SKView，是一个包含Sprite Kit场景的视图.

这里你实现了viewDidLoad()去创建一个启动时的GameScene，尺寸与视图本身相同。这就是最初的设置，现在屏幕上有些东西了！

添加一个精灵

首先下载这个功能的资源文件拖进你的Xcode项目中。确保Copy items into destination group's folder (if needed)这一项被选中，并且是在你SpriteKitSimpleGame target被选中的情况下。

下面打开GameScene.swift用如下代码替换内容:

```
import SpriteKit

class GameScene: SKScene {

  // 1
  let player = SKSpriteNode(imageNamed: "player")

  override func didMoveToView(view: SKView) {
    // 2
    backgroundColor = SKColor.whiteColor()
    // 3
    player.position = CGPoint(x: size.width * 0.1, y: size.height * 0.5)
    // 4
    addChild(player)
  }
}
```

让我们来逐步分析。

这里你为player(忍者)定义了一个私有常量，一个精灵的例子。如你所见创建一个精灵挺简单的，输入需要使用的图片名即可。

在Sprite Kit中设置场景的背景色如同设置其他背景色一样简单，这个你将其设为白色。

你将精灵定位在竖直方向10%的高度，水平中心处。

为了让精灵显示在场景中，你必须将其作为一个子项添加进场景中，与添加其它子视图的方法类似。

构建并运行下。女士们先生们，忍者闯进来了！

移动怪物

接下来你要在场景中添加一些怪物与你的忍者战斗。为了使事情变得有趣你需要移动这些怪物，否则的话就太简单了！让我来在屏幕右侧创建怪物，然后设定一个动作告诉它们向左移动。

在GameScene.swift中添加如下方法:

```
func random() -> CGFloat {
  return CGFloat(Float(arc4random()) / 0xFFFFFFFF)
}

func random(min min: CGFloat, max: CGFloat) -> CGFloat {
  return random() * (max - min) + min
}

func addMonster() {

  // Create sprite
  let monster = SKSpriteNode(imageNamed: "monster")

  // Determine where to spawn the monster along the Y axis
  let actualY = random(min: monster.size.height/2, max: size.height - monster.size.hei
ght/2)

  // Position the monster slightly off-screen along the right edge,
  // and along a random position along the Y axis as calculated above
  monster.position = CGPoint(x: size.width + monster.size.width/2, y: actualY)

  // Add the monster to the scene
  addChild(monster)

  // Determine speed of the monster
  let actualDuration = random(min: CGFloat(2.0), max: CGFloat(4.0))

  // Create the actions
  let actionMove = SKAction.moveTo(CGPoint(x: -monster.size.width/2, y: actualY), dura
tion: NSTimeInterval(actualDuration))
  let actionMoveDone = SKAction.removeFromParent()
  monster.runAction(SKAction.sequence([actionMove, actionMoveDone]))

}
```

这里我会啰嗦一点，为了让事情更好理解。在第一部分应该将场景做成我们之前讨论的那样，对于你想创建对象的位置做一些计算，设置对象的位置，然后将其添加到场景中，像之前添加player那样。

这里的新要素就是添加动作。Sprite Kit提供了大量的内置动作，可以帮助你随时改变精灵的状态，比如移动动作、旋转动作、渐变动作、动画动作等等。这里你可以看到怪物的三种动作:

- **SKAction.moveTo(_:duration:)**: 使用这个动作使对象从屏幕外移动到左边，注意你可以设置移动的持续时间，你可以随机改变移动速度。
- **SKAction.removeFromParent()**: Sprite Kit有一个方便的动作就是从父类中移除一个节点，从场景中删除。当场景中的怪物不可见时你会使用这个动作去移除它，这时很重要的因为否则你会一直添加怪物直到消耗掉所有内存资源。

- **SKAction.sequence(_:)**: 队列动作允许你添加一个动作队列，依次执行。这样你就可以先执行"移动"动作，完成后执行"从父类移除"动作。

> 注意:这块代码中包含了一些辅助方法使用arc4random()产生一个范围内的随机数，满足了在这个游戏中的随机数产生需求，如果你想将这块更完善的话看看iOS 9中新的GameplayKit的随机数API。

还差最后一件事，你需要调用创建怪物的方法！为了好玩，我们来不断地一直产生怪物。

在didMoveToView()的最后添加如下代码:

```
runAction(SKAction.repeatActionForever(
  SKAction.sequence([
    SKAction.runBlock(addMonster),
    SKAction.waitForDuration(1.0)
  ])
))
```

这里你运行了一个动作队列，调用了一块block代码，然后等待一秒。一直重复执行这个队列。

搞定了！构建并运行，现在你应该会看到一些怪物在屏幕上活蹦乱跳了:

射出子弹

在这里需要忍者作出一些动作 - 来添加射击功能吧！可以使用多种方式来实现射击，在这个游戏中需要当用户点击屏幕时进行射击动作，在点击的方向射出一个子弹。

我想用一个“移动”动作来实现这个功能，在这之前需要你做一些数学上的工作。

这是因为“移动”动作需要你提供子弹行动的终点，你不能仅仅利用触摸点因为触摸点只表达了player射击时的方向。实际上你想实现的是让子弹移动时通过触摸点直到飞出屏幕。

下面这张图描绘了这个问题:

如你所见，针对触摸点与原始点之前有一个在两个方向上偏移x与y的小三角形。你需要使用一个相似的大三角形，这个三角形的一个顶点在屏幕边界上。

如果你有一些基本数学向量的方法（不如添加与减去向量）来调用的话对计算来说会很有帮助。然后Sprite Kit默认没有这些方法你需要自己去编写。

幸运的是由于方便的Swift操作符加载这些也是很容易实现的。在你的文件头添加这些函数，在GameScene类之前:

```
func + (left: CGPoint, right: CGPoint) -> CGPoint {
  return CGPoint(x: left.x + right.x, y: left.y + right.y)
}

func - (left: CGPoint, right: CGPoint) -> CGPoint {
  return CGPoint(x: left.x - right.x, y: left.y - right.y)
}

func * (point: CGPoint, scalar: CGFloat) -> CGPoint {
  return CGPoint(x: point.x * scalar, y: point.y * scalar)
}

func / (point: CGPoint, scalar: CGFloat) -> CGPoint {
  return CGPoint(x: point.x / scalar, y: point.y / scalar)
}

#if !(arch(x86_64) || arch(arm64))
func sqrt(a: CGFloat) -> CGFloat {
  return CGFloat(sqrtf(Float(a)))
}
#endif

extension CGPoint {
  func length() -> CGFloat {
    return sqrt(x*x + y*y)
  }

  func normalized() -> CGPoint {
    return self / length()
  }
}
```

这些是典型数学向量函数的实现。如果你对向量不太熟悉，很困惑发生了什么，来看看数学向量的快速入门。

下一步在文件中添加一个新方法:

```
override func touchesEnded(touches: Set<UITouch>, withEvent event: UIEvent?) {

  // 1 - Choose one of the touches to work with
  guard let touch = touches.first else {
    return
  }
  let touchLocation = touch.locationInNode(self)

  // 2 - Set up initial location of projectile
  let projectile = SKSpriteNode(imageNamed: "projectile")
  projectile.position = player.position

  // 3 - Determine offset of location to projectile
  let offset = touchLocation - projectile.position

  // 4 - Bail out if you are shooting down or backwards
  if (offset.x < 0) { return }

  // 5 - OK to add now - you've double checked position
  addChild(projectile)

  // 6 - Get the direction of where to shoot
  let direction = offset.normalized()

  // 7 - Make it shoot far enough to be guaranteed off screen
  let shootAmount = direction * 1000

  // 8 - Add the shoot amount to the current position
  let realDest = shootAmount + projectile.position

  // 9 - Create the actions
  let actionMove = SKAction.moveTo(realDest, duration: 2.0)
  let actionMoveDone = SKAction.removeFromParent()
  projectile.runAction(SKAction.sequence([actionMove, actionMoveDone]))

}
```

这儿包含了很多东西，让我们来逐步分析:

1. SpriteKit中一个很酷的事情就是它有一些UITouch的方法，例如locationInNode(*:)*与*previousLocationInNode(*:)，使你可以通过SKNode的坐标系统获得一个触摸点的坐标。在这里你使用它得到屏幕坐标系统中触摸点的位置。
2. 然后你在player起始位置创建了一个子弹，注意你还没有添加到场景中，因为需要先做一个理智的判断 - 这个游戏不允许忍者向身后射击。
3. 通过计算触摸点位置与位置的差值得到一个从当前位置到触摸点位置的向量。
4. 如果X值小于0则意味着player在向后射击，这是游戏里所不允许的(真正的忍者不会向后看！)，所以直接return跳出。
5. 否则就可以将子弹添加到视图中了。

6. 调用normalized()使offset归一化，这样方便构成一个相同方向上的向量，因为 1 * length = length。
7. 将你想要的射击方向的向量乘以1000，为何是1000?这个数值足够它飞出屏幕边界:)
8. 将shoot amount添加到当前位置以得到它在屏幕上运动的终点。
9. 最后，像之前那样创建moveTo(_:, duration:)与removeFromParent()动作。

构建并运行，现在你的忍者就可以对迎面而来的部队进行连续射击了！

碰撞检测与物理效果:概述

现在你可以将飞镖扔到任何地方，不过你的忍者应该想要击落一些东西，所以让我们来加一些代码来检测子弹与目标的碰撞吧。

Sprite Kit有一个优点就是自带物理引擎，不仅擅长模拟眞实运动，也擅长碰撞检测。

让我们来在游戏中设置Sprite Kit的物理引擎来决定当怪物与子弹碰撞时的效果，在顶层的代码中你需要做:

- 设置物理场。一个物理场(physics world)就是进行物理运算模拟的地方。场景中默认设置了一个，你可以想要设置一个新属性，比方说重力。
- 为每个精灵创建物理体。在Sprite Kit中，你可以为每个精灵分配一个形状来进行碰撞检测，并设置属性，这就叫做一个物理体(physics body)。注意物理体不需要是与精灵完全相同的形状。一般来说为了方便起见选取一个近似的形状而不是每个像素都一样的形状，对于大多数游戏与移动这样做足够了。
- 为每个精灵类型设置类别。类别(category)是你可以设置的一个物理体属性，它用一个掩码来表示所属的群组。在这个游戏中，你需要有两个category，子弹的和怪物的。当两个物理体碰撞时，通过类别你可以知道需要处理的是哪类精灵。
- 设置一个判断接触的委托(contact delegate)方法来监测两个物理体是否碰撞。需要写一些代码去检测物体的类别，如果它们是怪物与子弹就让它们爆炸！

现在你了解了战斗计划，是时候添加进动作了！

碰撞检测与物理效果:实现

在GameScene.swift的头部添加如下结构体:

```
struct PhysicsCategory {
  static let None      : UInt32 = 0
  static let All       : UInt32 = UInt32.max
  static let Monster   : UInt32 = 0b1       // 1
  static let Projectile: UInt32 = 0b10      // 2
}
```

设置一些物理类型的常量。

> 注意:你也许奇怪这些语句有啥用。注意Sprite Kit中的类别仅仅是一个32位的整型数，相当于掩码。用32位整型数表示类别是一个很美妙的方法。这里你设置第一位表示一个怪物，下一位表示一个子弹。

下面让GameScene实现**SKPhysicsContactDelegate**协议:

```
class GameScene: SKScene, SKPhysicsContactDelegate {
```

Then inside add these lines after adding the player to the scene: 在**didMoveToView(_:)**方法中添加player到视图的代码后添加如下行:

```
physicsWorld.gravity = CGVectorMake(0, 0)
physicsWorld.contactDelegate = self
```

它设置了一个零重力场，并且设置了场景的委托，用来监测两个物理体的碰撞。

在**addMonster()**方法中，在创建怪物精灵的代码后面添加如下行:

```
monster.physicsBody = SKPhysicsBody(rectangleOfSize: monster.size) // 1
monster.physicsBody?.dynamic = true // 2
monster.physicsBody?.categoryBitMask = PhysicsCategory.Monster // 3
monster.physicsBody?.contactTestBitMask = PhysicsCategory.Projectile // 4
monster.physicsBody?.collisionBitMask = PhysicsCategory.None // 5
```

让我们来逐行分析下:

1. 为精灵创建一个物理体。在这种情况下，物理体的形状被定义为与精灵相同尺寸的矩形，因为这是个合适的怪物近似形状。
2. 设置精灵为动态(dynamic)的。这意味着物理引擎将不会控制怪物的移动，需要通过你已经写过的代码来进行（使用移动动作）。
3. 设置categoryBitMask为之前你定义的怪物类别。
4. contactTestBitMask的作用是设置当与另一个目标接触时需要提醒的目标类别，这里设置成子弹。
5. collisionBitMask的作用是设置与另一个物体接触时需要用物理引擎处理做出响应(比方说

反弹)的类别。 你不会想让怪物与子弹互相反弹的，在这个游戏中可以让它们穿过彼此，因此设为空。

接下来在touchesEnded(_:withEvent:)中添加一些类似的代码，在设置子弹位置的代码后:

```
projectile.physicsBody = SKPhysicsBody(circleOfRadius: projectile.size.width/2)
projectile.physicsBody?.dynamic = true
projectile.physicsBody?.categoryBitMask = PhysicsCategory.Projectile
projectile.physicsBody?.contactTestBitMask = PhysicsCategory.Monster
projectile.physicsBody?.collisionBitMask = PhysicsCategory.None
projectile.physicsBody?.usesPreciseCollisionDetection = true
```

试试看，自己能否能否理解每一行代码，如果不能再去看看上面的解析吧。

接下来，添加一个当子弹与怪物碰撞时调用的方法。注意它不会自动调用，你如果在后面调用它。

```
func projectileDidCollideWithMonster(projectile:SKSpriteNode, monster:SKSpriteNode) {
  print("Hit")
  projectile.removeFromParent()
  monster.removeFromParent()
}
```

你要做的就是当子弹与怪物碰撞时将它们从场景中移除。挺简单的，不是吗？

现在是时候来实现接触的代理方法了，在文件中添加如下的新方法:

```
func didBeginContact(contact: SKPhysicsContact) {

  // 1
  var firstBody: SKPhysicsBody
  var secondBody: SKPhysicsBody
  if contact.bodyA.categoryBitMask < contact.bodyB.categoryBitMask {
    firstBody = contact.bodyA
    secondBody = contact.bodyB
  } else {
    firstBody = contact.bodyB
    secondBody = contact.bodyA
  }

  // 2
  if ((firstBody.categoryBitMask & PhysicsCategory.Monster != 0) &&
      (secondBody.categoryBitMask & PhysicsCategory.Projectile != 0)) {
    projectileDidCollideWithMonster(firstBody.node as! SKSpriteNode, monster: secondBo
dy.node as! SKSpriteNode)
  }

}
```

在之前你在场景中设置了物理场的接触委托，这个方法会在两个物理体碰撞时被调用。

在这个方法中有两部分：

1. 在这个方法中你传递了两个碰撞的物理体，没有规定传递它们的顺序，因此这段代码利用它们的categoryBitMask属性来进行排序这样你就在后面做出假设。.
2. 在最后检查一下两个碰撞体是否是子弹和怪物，如果是的话调用你之前写的那个方法。构建并运行一下，现在当你的子弹碰到目标时会一起消失！

完成触摸

你离一个可运行(不过还是很简陋)的游戏已经很近了。需要添加一些效果与音乐(游戏总要有点声音吧！)还有一些游戏逻辑。

在项目中已经有一些现成的很酷的背景乐与一个很棒的射击音效，你只需要播放它们即可！

在didMoveToView(_:)的最后添加如下代码:

```
let backgroundMusic = SKAudioNode(fileNamed: "background-music-aac.caf")
backgroundMusic.autoplayLooped = true
addChild(backgroundMusic)
```

这个使用了SKAudioNode —— 一个iOS 9中的新类，在你的游戏中播放与循环背景乐。

在touchesEnded(_:withEvent:)中添加如下这行来添加音效:

```
runAction(SKAction.playSoundFileNamed("pew-pew-lei.caf", waitForCompletion: false))
```

非常方便，不是吗？你可以用一行来播放音效！

构建并运行，享受绝妙的旋律吧！

> 注意：如果你没听到背景乐，试试在真机上测试。

哥们！游戏结束！

现在让我们来创建一个新场景，用来显示“你赢了”或“你输了”。创建一个名为GameOverScene的新swift文件。

用如下代码替代GameOverScene.swift中的内容：

```
import Foundation
import SpriteKit

class GameOverScene: SKScene {

  init(size: CGSize, won:Bool) {

    super.init(size: size)

    // 1
    backgroundColor = SKColor.whiteColor()

    // 2
    let message = won ? "You Won!" : "You Lose :["

    // 3
    let label = SKLabelNode(fontNamed: "Chalkduster")
    label.text = message
    label.fontSize = 40
    label.fontColor = SKColor.blackColor()
    label.position = CGPoint(x: size.width/2, y: size.height/2)
    addChild(label)

    // 4
    runAction(SKAction.sequence([
      SKAction.waitForDuration(3.0),
      SKAction.runBlock() {
        // 5
        let reveal = SKTransition.flipHorizontalWithDuration(0.5)
        let scene = GameScene(size: size)
        self.view?.presentScene(scene, transition:reveal)
      }
    ]))

  }

  // 6
  required init(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
  }
}
```

这里有几个需要注意的点：

1. 设置背景色为白色，像之前你对主场景做的那样。
2. 根据won参数来设置消息是“You Won” 还是 “You Lose”（用了三目运算符）。
3. 这就是使用Sprite Kit怎样将一个标签显示到屏幕中的方法，如你所见，这是很简单的 - 你只需要选择你的字体并设定一个新的参数即可。
4. 最后设置并运行一个包含两个动作的队列。我将使用一行来向你展示它是多么的方便(而不是使用两个分离的单独动作变量)。一开始西安等待3秒，然后使用runBlock动作去执行

剩下的代码。

5. 这就是如何在Sprite Kit中跳转到另一个场景。首先你要在不同的动画变换中选择出你先要显示出场景的方法 - 你选择了一个持续0.5秒的反转动画。接着你创建了想要显示的场景，在self.view属性上使用presentScene(_:transition:)方法。
6. 如果你要重载一个场景的初始化，你必须实现required init(coder:)这个初始化方法。然而这个初始化永远不会被调用，需要添加一个fatalError(_:)方法来实现。

到现在为止还算顺利，还需要设置加载游戏时的主要场景。

返回GameScene.swift中，在addMonster()方法中, 用下面的代码替换最后一行的怪物动作：

```
let loseAction = SKAction.runBlock() {
  let reveal = SKTransition.flipHorizontalWithDuration(0.5)
  let gameOverScene = GameOverScene(size: self.size, won: false)
  self.view?.presentScene(gameOverScene, transition: reveal)
}
monster.runAction(SKAction.sequence([actionMove, loseAction, actionMoveDone]))
```

这里创建了一个“失败动作”，当一个怪物飞出屏幕时会显示game over画面。看看你是否理解每一行，如果没有请看看之前的解析。

现在你应该处理一下胜利时的情况，不要对你的玩家太残酷！在GameScene的顶部添加一个新属性，在player的声明之后：

```
var monstersDestroyed = 0
```

在projectile(_:didCollideWithMonster:)上添加如下代码：

```
monstersDestroyed++
if (monstersDestroyed > 30) {
  let reveal = SKTransition.flipHorizontalWithDuration(0.5)
  let gameOverScene = GameOverScene(size: self.size, won: true)
  self.view?.presentScene(gameOverScene, transition: reveal)
}
```

直接运行一下，现在有了胜利与失败的条件并能在适当的时候看到game over画面了。

OAuth 2.0 with Swift Tutorial

Swift的OAuth 2.0使用指南

> - 原文链接 : OAuth 2.0 with Swift Tutorial
> - 原文作者 : Corinne Krych
> - 译者 : yrq110

在你将内容分享到喜爱的社交网络(Facebook, Twitter等)上或者你企业的OAuth2服务器上时，很可能不经意使用了OAuth2与其他种类的协议，你甚至不知道内部的原理。 不过你知道如何使用OAuth2连接你的服务与iOS app吗？

在这篇教程中，你会使用一个名为Incognito的分享型app来学习如何使用AeroGear OAuth2, AFOAuth2Manager和OAuthSwift这些开源OAuth2库来将你的资源分享到GoogleDrive上。

目录

入门

下载Incognito开始工程。开始工程使用CocoaPods来安装AeroGear与其他你需要的一切，包括生成pods与xcworkspace目录。

在Xcode中打开Incognito.xcworkspace。工程基于Xcode的单视图应用模板，有包含单一ViewController.swift视图控制器的故事版，所有UI与动作在ViewController.swift中都已经处理好了。

构建并运行你的工程，会看到如下图这样：

这个app可以让你选择一张自拍然后在图片上加一些装扮，你认出乔装后的我了吗 :]

> 注意：在模拟器中添加照片，可以按Cmd + Shift + H回到home界面然后将你的图片拖到模拟器中。

所有需要在app中添加的东西就是使用了3个不同的OAuth2库来完成分享到Google Drive的特性。

不可能的任务? 不，没有你搞不定的:]

让我来给你讲个故事吧，而非让你看RFC6749 OAuth2的规范，那样太无聊了。

讲讲为何需要OAuth2

周一的清晨，我们的开发小哥Bob在咖啡机前碰到了友好的极客Alice。Bob拿着一大摞文件，好像挺忙的样子，他的老板想让他研究一下OAuth2规范并加入到Incognito应用中。

当然，如果你把俩开发者放到一个咖啡厅中他们会开始聊聊比较极客的话题。Bob问Alice：

“我们要用OAuth2解决哪些问题？”

一方面，你可以使用API形式的服务，比如使用Twitter的API你可以得到跟随者或推文的列表。这些API处理你的在登录与密码的保护下的加密数据。

另一方面，你有使用这些服务的app。那些app会访问你的数据，不过你会信任所有拥有你证书的服务吗？也许会，也许不会。

这就带来了委托访问的概念。OAuth2让用户授权第三方的app在不提供密码的情况下通过安全令牌来访问web资源。令牌中不会包含密码信息，这意味着你的密码安全的存放在主服务器中，每一个想要连接服务器的app都会得到专用令牌。之后如果你想撤销app访问的话可以撤销令牌。

OAuth2有以下四个要素：

- 授权服务器: 响应认证与授权，提供访问令牌

- 资源服务器: 对有效的令牌提供资源服务
- 资源用户: 数据的拥有者 - Incognito的终端用户
- 客户端: Incognito移动端app

OAuth2规范使用授权流(grant flows)来描述交互行为。

规范中定义了四种不同的授权流，可以被归为两类：

- 3-legged流：这种情况下终端用户需要获得授权权限。简化授权是为了那些不方便存储安全令牌的使用浏览器的app。授权码授权，会生成一个访问令牌与随机生成的令牌，使客户端能安全的保存令牌。在这样的移动app的客户端中会有一个安全存放令牌的地方，类似iOS中的钥匙串。
- 2-legged流: 凭证会直接发给app。不同之处在于资源用户直接在客户端中输入了凭证，比如说你作为一个开发者在访问像Parse等的API时会将密钥存放在你的app中。

在这篇教程中会使用已有的Google Drive账号，并上传Incognito自拍。这种情况下使用3-legged认证比较好。

授权之舞

虽然使用开源库时会隐藏那些OAuth2协议中棘手的细节，不过了解一下基本的工作原理会有助于你更好的进行配置。

这里是授权认证之舞的步骤：

Step 0: 注册

app需要注册需要访问的服务。对你来说，Incognito需要注册Google Drive。别担心，之后会介绍具体怎么做。

Step 1: 授权

舞蹈开始时Incognito会给第三方服务端发一个请求的授权码，包含以下信息：

- 客户端ID: 在注册服务阶段提供。定义与服务会话的app。
- 客户端密钥: 在注册服务阶段提供。一个服务与app之前的密钥，打包进了app的二进制文件。
- 重定向URI: 在用户使用证书访问服务后重定向到的位置，并授权许可。
- 作用域: 用来告诉服务app所拥有的许可级别。

app之后会切换到浏览器界面。一旦用户登入，则Google授权服务器会显示一个授权页: “Incognito想要访问你的相册：允许/拒绝”，当终端用户点击“允许”时，服务器会使用重定向URI跳转回Incognito中，并给app发送一个授权码。

Step 2: 交换授权码与令牌

授权码是临时的，因此OAuth2库需要用临时码交换得到一个合适的访问令牌，是随机生成的。

Step 3: 取得资源

使用访问令牌，Incognito可以访问服务器上受保护的资源-用户所授权访问的资源，可以自由的上传图片。

准备好实践了吗？首先，你需要注册OAuth2的提供商-Google。

注册OAuth2提供商

如果你没有Google账户，现在就去创建一个吧。我们会等你:]

使用你的浏览器打开http://console.developer.google.com 进行操作

点击创建工程并起名为Incognito：

接着，你需要启用API。

进入APIs&auth\APIs界面，然后点击Google Apps APIs\Drive API。在接下来的视图中点击启用API：

现在你需要创建一个新的证书来访问app中的账户。

进入APIs&auth\Credentials界面点击创建新的客户端ID按钮，点击同意在出现的视图中填写以下信息：

- 邮箱: 选择你的邮箱地址
- 项目名称: Incognito
- 主页: http://www.raywenderlich.com

点击保存，回到客户端ID界面。

选择安装的应用，然后选择iOS输入com.raywenderlich.Incognito作为你的Bundle ID。

授权服务器将会使用这个bundle id作为重定向URI进入app。

最后点击创建客户端ID，最后的界面会显示生成的客户端ID、客户端密钥与重定向URI：

现在成功注册了Google服务，可以开始使用第一个OAuth2库来实现OAuth2了：依赖外部浏览器的AeroGear。

使用AeroGear与外部浏览器授权

打开ViewController.swift在文件头部添加如下代码：

```
import AeroGearHttp
import AeroGearOAuth2
```

现在，在ViewController类中添加实例变量：

```
var http: Http!
```

在viewDidLoad()中初始化它：

```
self.http = Http()
```

使用AerGearHttp库中的这个Http的实例来执行HTTP的请求。

在ViewController.swift中找到空的share(:)方法并添加如下代码：

```
let googleConfig = GoogleConfig(
clientId: "YOUR_GOOGLE_CLIENT_ID",                          // [1] Define a Googl
e configuration
scopes:["https://www.googleapis.com/auth/drive"])           // [2] Specify scope

let gdModule = AccountManager.addGoogleAccount(googleConfig)     // [3] Add it to Acco
untManager
self.http.authzModule = gdModule                            // [4] Inject the Aut
hzModule
                                                            // into the HTTP laye
r object

let multipartData = MultiPartData(data: self.snapshot(),         // [5] Define multi-p
art
          name: "image",
          filename: "incognito_photo",
          mimeType: "image/jpg")
let multipartArray =  ["file": multipartData]

self.http.POST("https://www.googleapis.com/upload/drive/v2/files",   // [6] Upload ima
ge
               parameters: multipartArray,
               completionHandler: {(response, error) in
  if (error != nil) {
    self.presentAlert("Error", message: error!.localizedDescription)
  } else {
    self.presentAlert("Success", message: "Successfully uploaded!")
  }
})
```

上面的方法进行了如下操作：

1. 需要使用你的Google控制台中的客户端ID来替换上面代码中的YOUR_GOOGLE_CLIENT_ID，进行正确的配置。
2. 定义了授权请求的域名，在Incognito中你需要访问drive API。
3. 使用AccountManager的方法进行OAuth模块的初始化。
4. 接着将OAuth2模块导入连接着授权模块的HTTP对象中。
5. 创建一个包含多种数据的对象来封装你想发送给服务器的信息。
6. 最后调用一个简单的HTTP请求上传图片，POST()会检查OAuth2模块是否加进了HTTP中，并产生一个回调：
    - 若没有访问令牌存在则开始授权码的授权
    - 若需要则重新生成访问令牌
    - 若所有令牌都可用，则仅仅调用POST指令

> 注意：想了解有关AeroGear OAuth2的信息的话可以看看AeroGear的在线文档与API手册，或者看看Pods中的源代码。

构建并运行app，选择一张图片，添加一些覆盖物，然后点击分享按钮，会提示你输入Google证书，如果你已经登录过了，你的整数会存放在缓存中。重定向到认证界面，点击接受，然后...

Boom沙卡拉卡 你收到了一条Safari无法打开页面的错误信息。发生了什么？

当你点击接受后，Google OAuth站点会重定向到com.raywenderlich.Incognito://[some url]。因此需要设置app使其能打开这个 URL scheme。

> 注意：Safari将你的授权响应信息存放在了模拟器的cookie中，所以别立即就重新授权，为了清空模拟器的这些cookie需要重置iOS模拟器的内容与设置。

配置URL Scheme

允许你的用户重定向回到Incognito中，你需要自定义一个关联的URL scheme。 在Info.plist.文件中点击右键选择Open As\Source Code，在plist的底部，</dict>标签的右边加入如下代码：

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>com.raywenderlich.Incognito</string>
        </array>
    </dict>
</array>
```

scheme是一个URL中的第一个部分，在网页中的scheme通常是http或https。iOS app可以自定义它们自己的 URL schemes，比如com.raywenderlich.Incognito://doStuff，重点就是自定义一个独特的scheme使其与设备中安装的其他app有所区别。

在OAuth2之舞中使用你自定义的URL scheme根据请求来返回到app中。自定义Scheme中包含一些参数，在这里，授权码被包含在code参数中。 OAuth2库会从URL中提取授权码然后在下一次请求时与访问令牌进行交换。

你需要在Incognito的AppDelegate类中实现一个基于自定义URL scheme的响应方法。

打开AppDelegate.swift，在文件头部添加如下代码:

```
import AeroGearOAuth2
```

接着像下面这样实现application(_: openURL: sourceApplication: annotation)：

```
func application(application: UIApplication,
  openURL url: NSURL,
  sourceApplication: String?,
  annotation: AnyObject?) -> Bool {
    let notification = NSNotification(name: AGAppLaunchedWithURLNotification,
      object:nil,
      userInfo:[UIApplicationLaunchOptionsURLKey:url])
    NSNotificationCenter.defaultCenter().postNotification(notification)
    return true
}
```

这个方法中创建了一个包含打开app时URL信息的NSNotification。AeroGearOAuth2库会接收到这个通知，然后调用你之前插入的POST方法中的completionHandler。

构建并运行一下工程，选个时髦的自拍并打扮一下，点击分享按钮，认证，观察一下发生了什么：

你可以从这里下载完成后的应用。

在OAuth2授权过程中跳转到外部浏览器未免显得太笨重，应该有一个简化的实现方法...

使用嵌入的Web视图

嵌入的web视图有更好的用户体验。使用UIWebView来实现而不是跳转到Safari中。从安全的角度来讲，使用app的代码来处理登录表单与提供商的数据会使安全性降低。当用户输入信息时app可以使用JS去访问用户证书，如果你的终端用户信任app的安全性的话是一个可考虑的方案。

来使用OAuthSwift库重新实现分享的方法，不过这次使用的是嵌入的web视图。

使用嵌入web视图的OAuthSwift

将会开始一个不同的新工程，所以先关闭Xcode的工作区，下载Incognito初始工程的版本，用Xcode打开Incognito.xcworkspace文件。

构建并运行工程，看起来挺熟悉的。

首先需要将OAuthSwift库加入到工程中。

打开ViewController.swift在文件头部添加如下代码：

```
import OAuthSwift
```

在share()方法中添加如下代码:

```
// 1 Create OAuth2Swift object
let oauthswift = OAuth2Swift(
  consumerKey:    "YOUR_GOOGLE_DRIVE_CLIENT_ID",         // 2 Enter google app settings

  consumerSecret: "YOUR_GOOGLE_DRIVE_CLIENT_SECRET",
  authorizeUrl:   "https://accounts.google.com/o/oauth2/auth",
  accessTokenUrl: "https://accounts.google.com/o/oauth2/token",
  responseType:   "code"
)
// 3 Trigger OAuth2 dance
oauthswift.authorizeWithCallbackURL(
  NSURL(string: "com.raywenderlich.Incognito:/oauth2Callback")!,
  scope: "https://www.googleapis.com/auth/drive",        // 4 Scope
  state: "",
  success: { credential, response in
    var parameters =  [String: AnyObject]()
    // 5 Get the embedded http layer and upload
    oauthswift.client.postImage(
      "https://www.googleapis.com/upload/drive/v2/files",
      parameters: parameters,
      image: self.snapshot(),
      success: { data, response in
        let jsonDict: AnyObject! = NSJSONSerialization.JSONObjectWithData(data,
          options: nil,
          error: nil)
        self.presentAlert("Success", message: "Successfully uploaded!")
      }, failure: {(error:NSError!) -> Void in
        self.presentAlert("Error", message: error!.localizedDescription)
    })
  }, failure: {(error:NSError!) -> Void in
    self.presentAlert("Error", message: error!.localizedDescription)
})
```

上面的代码进行了如下操作：

1. 首先创建一个OAuth2Swift对象，将为你执行OAuth之舞。
2. 将YOUR_GOOGLE_CLIENT_ID与YOUR_GOOGLE_DRIVE_CLIENT_SECRET替换为你的客户端ID与客户端密钥。
3. 通过oauthswift实例来请求授权。
4. 这个域名是你请求访问的Drive API。
5. 如果授权成功则可以开始上传图片了。

配置URL

如之前的工程那样的，需要设置一个Incognito接受的URL scheme，你所要做的就是实现处理自定义URL的代码。

打开AppDelegate.swift添加如下代码：

```
import OAuthSwift
```

然后像下面这样实现application(_:openURL: sourceApplication: annotation:)方法:

```
func application(application: UIApplication,
  openURL url: NSURL,
  sourceApplication: String?,
  annotation: AnyObject?) -> Bool {
    OAuth2Swift.handleOpenURL(url)
    return true
}
```

跟AeroGearOAuth2不同, OAuthSwift使用一个类方法去处理并解析返回的URL。不过如果你看看handleOpenURL(_) 方法的代码时会发现仅仅是发送了一个NSNotification，就像用AeroGearOAuth2时需要你做的那样!

构建并运行你的工程，创建一个新的自拍然后上传。喔! 再一次成功了! 挺简单的吧 :]

显示UIWebView

现在就要如约添加web view了。点击Incognito文件夹，在Xcode中的工程导航栏处选择File\New\File，然后选择iOS\Source\Swift File，将其命名为 WebViewController然后保存进工程中。

然后打开WebViewController.swift添加如下代码：

```
import UIKit
import OAuthSwift

class WebViewController: OAuthWebViewController {
  var targetURL : NSURL?
  var webView : UIWebView = UIWebView()

  override func viewDidLoad() {
    super.viewDidLoad()
    webView.frame = view.bounds
    webView.autoresizingMask =
      UIViewAutoresizing.FlexibleHeight | UIViewAutoresizing.FlexibleHeight
    webView.scalesPageToFit = true
    view.addSubview(webView)
    loadAddressURL()
  }

  override func setUrl(url: NSURL) {
    targetURL = url
  }

  func loadAddressURL() {
    if let targetURL = targetURL {
      let req = NSURLRequest(URL: targetURL)
      webView.loadRequest(req)
    }
  }
}
```

在上面的代码中创建了一个继承OAuthWebViewController的WebViewController，这个类只实现了一个方法：SetUrl:。在viewDidLoad()中调整web view的尺寸然后添加到viewcontroller的父视图中。此外，在OAuth2Swift的实例中加载URL，创建一个request。

接着打开ViewController.swift找到share()方法。在创建完oauthswift实例后添加如下代码：

```
oauthswift.webViewController = WebViewController()
```

这里会告诉oauthswift实例使用刚刚创建的web view controller。

最后，打开AppDelegate.swift，修改application(_:openURL: sourceApplication: annotation:)方法，在return true的前面添加如下代码：

```
// [1] Dismiss webview once url is passed to extract authorization code
UIApplication.sharedApplication().keyWindow?.rootViewController?.dismissViewController
Animated(true, completion: nil)
```

构建并运行工程，注意当授权窗口出现时，并不是通过Safari显示的，未产生app间的切换。由于默认在你app中是不会存储cookies的因此每次都会出现授权的窗口。

使用一个UIWebView去授权Google当然会更流畅，的确是! :]

你可以在这里下载最终的Incognito。

在这个教程中还剩下一件事。来重新审视share()方法，使用著名的HTTP库AFNetworking在OAuth2中的应用。

使用AFOAuth2Manager授权

AFOAuth2Manager使用一种与其他OAuth2库完全不同的方式：使用基于著名的AFNetworking框架的底层API。至于你想用一个UIWebView还是打开一个外部的浏览器完全由你来决定，开发者可自由选择在OAuth2之舞中第1步的初始化方式。

这个部分的教程从另一个初始工程开始，关闭已有的工程下载这个新的：Incognito starter project.

打开一个新工程进入ViewController.swift中，首先定义一些帮助方法与扩展。

在文件头部添加如下String扩展：

```
extension String {
  public func urlEncode() -> String {
    let encodedURL = CFURLCreateStringByAddingPercentEscapes(
          nil,
      self as NSString,
      nil,
      "!@#$%&*'();:=+,/?[]",
      CFStringBuiltInEncodings.UTF8.rawValue)
    return encodedURL as String
  }
}
```

上面的代码扩展了一个String类的函数，使用URL编码一段字符串。

在ViewController中添加如下方法：

```
func parametersFromQueryString(queryString: String?) -> [String: String] {
  var parameters = [String: String]()
  if (queryString != nil) {
    var parameterScanner: NSScanner = NSScanner(string: queryString!)
    var name:NSString? = nil
    var value:NSString? = nil
    while (parameterScanner.atEnd != true) {
      name = nil;
      parameterScanner.scanUpToString("=", intoString: &name)
      parameterScanner.scanString("=", intoString:nil)
      value = nil
      parameterScanner.scanUpToString("&", intoString:&value)
      parameterScanner.scanString("&", intoString:nil)
      if (name != nil && value != nil) {
        parameters[name!.stringByReplacingPercentEscapesUsingEncoding(NSUTF8StringEnco
ding)!]
          = value!.stringByReplacingPercentEscapesUsingEncoding(NSUTF8StringEncoding)
      }
    }
  }
  return parameters
}
```

从URL字符串中提取中查询的参数字符串。比方说如果查询字符串是name=Bob&age=21则方法会返回一个字典：name => Bob, age => 21。

接着你需要在ViewController中定义一个辅助函数，用来提取NSNotification中URL的OAuth码。

在share()方法下面添加如下方法:

```
func extractCode(notification: NSNotification) -> String? {
  let url: NSURL? = (notification.userInfo as!
    [String: AnyObject])[UIApplicationLaunchOptionsURLKey] as? NSURL

  // [1] extract the code from the URL
  return self.parametersFromQueryString(url?.query)["code"]
}
```

使用刚才实现的方法从查询字符串字典中抓取了键"code"的值。

在share()中添加如下:

```
// 1 Replace with client id /secret
let clientID = "YOUR_GOOGLE_CLIENT_ID"
let clientSecret = "YOUR_GOOGLE_CLIENT_SECRET"

let baseURL = NSURL(string: "https://accounts.google.com")
let scope = "https://www.googleapis.com/auth/drive".urlEncode()
```

```
let redirect_uri = "com.raywenderlich.Incognito:/oauth2Callback"

if !isObserved {
  // 2 Add observer
  var applicationLaunchNotificationObserver = NSNotificationCenter.defaultCenter().add
ObserverForName(
    "AGAppLaunchedWithURLNotification",
    object: nil,
    queue: nil,
    usingBlock: { (notification: NSNotification!) -> Void in
      // [5] extract code
      let code = self.extractCode(notification)

      // [6] carry on oauth2 code auth grant flow with AFOAuth2Manager
      var manager = AFOAuth2Manager(baseURL: baseURL,
        clientID: clientID,
        secret: clientSecret)
      manager.useHTTPBasicAuthentication = false

      // [7] exchange authorization code for access token
      manager.authenticateUsingOAuthWithURLString("o/oauth2/token",
        code: code,
        redirectURI: redirect_uri,
        success: { (cred: AFOAuthCredential!) -> Void in

          // [8] Set credential in header
          manager.requestSerializer.setValue("Bearer \(cred.accessToken)",
            forHTTPHeaderField: "Authorization")

          // [9] upload photo
          manager.POST("https://www.googleapis.com/upload/drive/v2/files",
            parameters: nil,
            constructingBodyWithBlock: { (form: AFMultipartFormData!) -> Void in
              form.appendPartWithFileData(self.snapshot(),
                name:"name",
                fileName:"fileName",
                mimeType:"image/jpeg")
            }, success: { (op:AFHTTPRequestOperation!, obj:AnyObject!) -> Void in
              self.presentAlert("Success", message: "Successfully uploaded!")
            }, failure: { (op: AFHTTPRequestOperation!, error: NSError!) -> Void in
              self.presentAlert("Error", message: error!.localizedDescription)
          })
        }) { (error: NSError!) -> Void in
          self.presentAlert("Error", message: error!.localizedDescription)
      }
  })
  isObserved = true
}

// 3 calculate final url
var params = "?scope=\(scope)&redirect_uri=\(redirect_uri)&client_id=\(clientID)&respo
nse_type=code"
// 4 open an external browser
```

```
UIApplication.sharedApplication().openURL(NSURL(string: "https://accounts.google.com/o
/oauth2/auth\(params)")!)
```

哇哦，这个方法好长！如果一步步分析的话就会理解的：

1. 通常，你需要使用Google控制台的客户端id与客户端密钥替换掉YOUR_GOOGLE_CLIENT_ID与YOUR_GOOGLE_DRIVE_CLIENT_SECRET
2. 接着添加一个notification的观察者
3. 创建一个OAuth所需的参数列表
4. 使用Safari打开URL开始OAuth之舞
5. 一旦用户授权成功，这个闭包就会执行，开始从回调URL中提取OAuth码
6. OAuth流的第二步：交换OAuth码得到令牌
7. 一得到令牌...
8. ...就绑定到HTTP头中...
9. ...最后, 上传图片至Google Drive

还差一步!

完成上面的操作后，打开AppDelegate.swift，添加如下方法：

```
func application(application: UIApplication,
  openURL url: NSURL,
  sourceApplication: String?,
  annotation: AnyObject?) -> Bool {
    let notification = NSNotification(
      name: "AGAppLaunchedWithURLNotification",
      object:nil,
      userInfo:[UIApplicationLaunchOptionsURLKey:url])
    NSNotificationCenter.defaultCenter().postNotification(notification)
    return true
}
```

启动了一个监听授权码的notification，目的是为了获得URL中的授权码。

最后一次构建并运行工程，再次成功运行! 效果如之前一样不过这次使用了AFOAuth2Manager!

你可以在这里下载最终的Incognito AFOAuth2Manager工程。

Unit Testing in Xcode 7 with Swift

使用Swift在Xcode7中的单元测试

> - 原文链接：Unit Testing in Xcode 7 with Swift
> - 原文作者：MAXIME DEFAUW
> - 译者：yrq110

每个iOS开发者都需要时不时测试他们的app。如果你不是某种狂热的码农，你应该体会过那种找了好几个小时的bug结果原因出在一个很简单的语法错误时的绝望的感觉，如果没有找到bug那将会更糟。不论你是初学者还是开发过很多app的老手，有规划的进行单元测试都会使你的代码有更高的可信度、更安全，也更容易的debug！

幸运的是Xcode7与Siwft都支持单元测试。虽然单元测试不意味着app没有bug，不过这依旧是一种非常有效的方法去证明代码的每一个部分或单元都尽职尽责，帮助debug过程。

顾名思义，在单元测试中会根据一个代码单元创建小巧且特定的功能性测试。如果通过了测试则会显示一个绿色log，如果由于某种原因没有成功，Xcode会给测试标记“failed”，可以通过这个标记来检视你的代码，去查找失败的原因。

目录

来看一个Demo

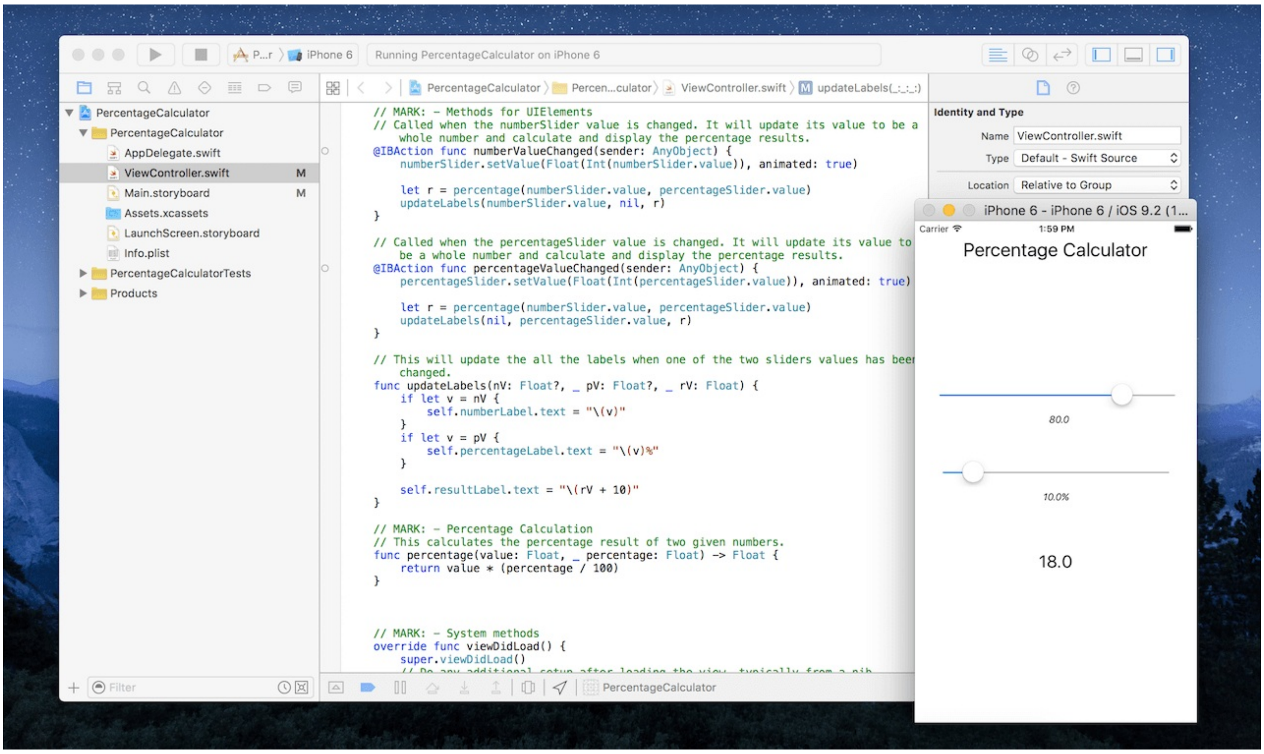
首先，下载这个为你准备的开始工程，一个计算数字百分比结果的计算器app (比如:80的10%是8)

这个百分比计算器工程很简单的，只需要看看ViewController.swift这个基本文件，代码的注释很直观。

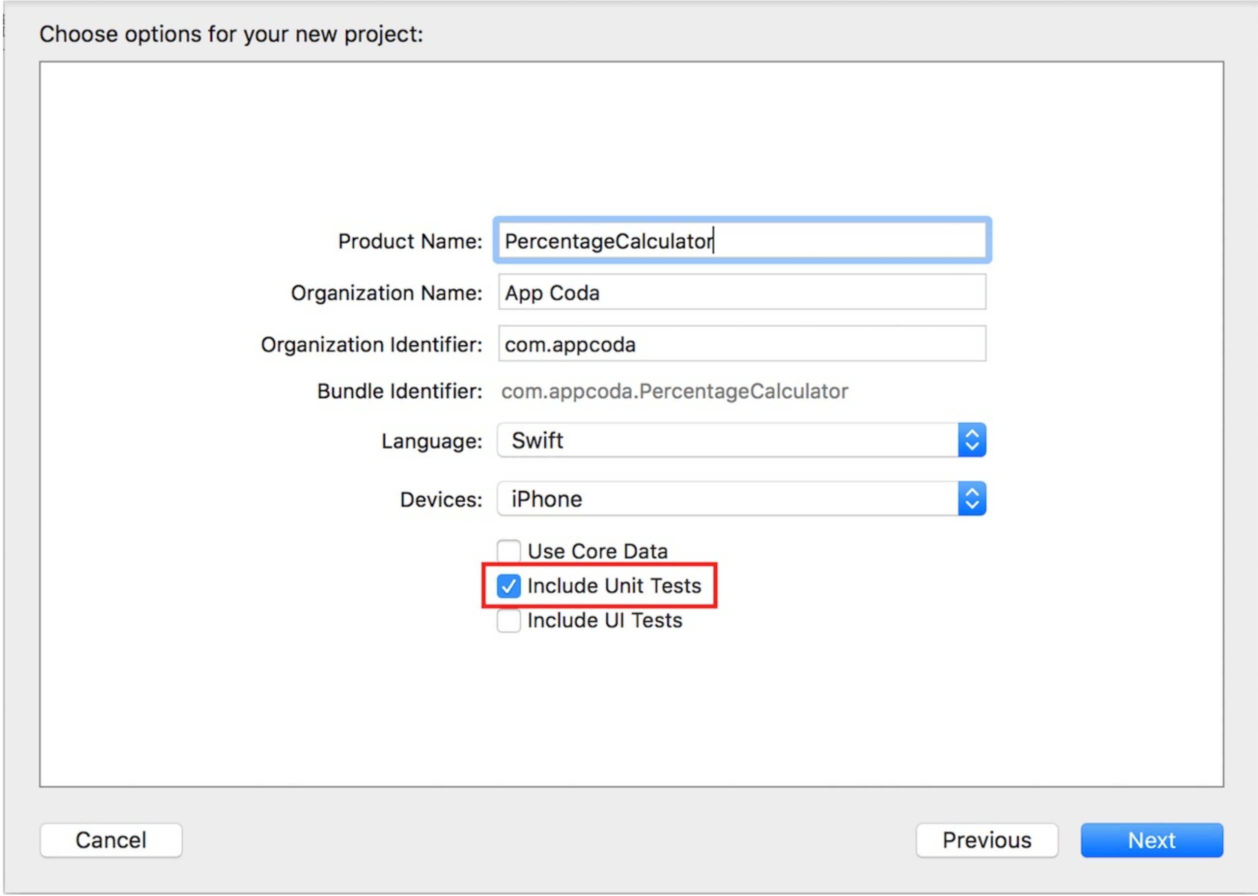
其中有5个IBOutlets: 对应除title以为的屏幕中的每一个控件, 与2个IBActions：分别对应两个滑条。 每个IBAction的名字准确的表达了这个方法的作用。若改变一个滑条的值时，则百分比或数字就会改变。

这里有2个简单的方法，“updateLabels()” 和 “percentage()”。第一个方法的作用的是当滑条的值改变时刷新label显示的内容，第二个的作用是接受两个浮点数然后返回百分比的计算结果。

在模拟器中运行app。一开始一切都很正常，不过如果你改变数字的话会注意到结果会出错。为了找出bug，我们可以将代码分为不同单元然后分别测试它们。这个过程不会解决bug，不过可以缩小你查找bug的范围。



在创建工程时默认选择包含一个测试文件。（如果你想手动添加，可以在选择 File > New > File > Unit Test Case Class under iOS Source）这时，Xcode已经帮你创建好了测试文件，在工程中的“PercentageCalculatorTests” 文件夹下。



在PercentageCalculatorTests.swift中的PercentageCalculatorTests class里有4个准备好的方法。其中有两个是示例程序，可以删除(可以通过test关键字看出是测试方法)。另外两个，setUp()与tearDown() 是特殊的样板方法，分别对应开始测试与结束测试时执行的方法。

开始编写单元测试

是时候来写出第一个单元测试方法了！在这篇教程中我们仅测试ViewController类，需要在PercentageCalculatorTests中添加一个实例。

```
class PercentageCalculatorTests: XCTestCase {
    var vc: ViewController!

    override func setUp() {
        super.setUp()
        // Put setup code here. This method is called before the invocation of each te
st method in the class.
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each
test method in the class.
        super.tearDown()
    }

}
```

PercentageCalculatorTests类是XCTestCase的一个子类，使用了XCTest框架进行打包，每一个XCTextCase子类的实例负责你工程中某一部分的测试，例如一个特性。

在setup方法中初始化vc，这样在setUp()调用后，在每一个测试方法中都能得到一个ViewController实例。 像这样修改setUp()方法 ：

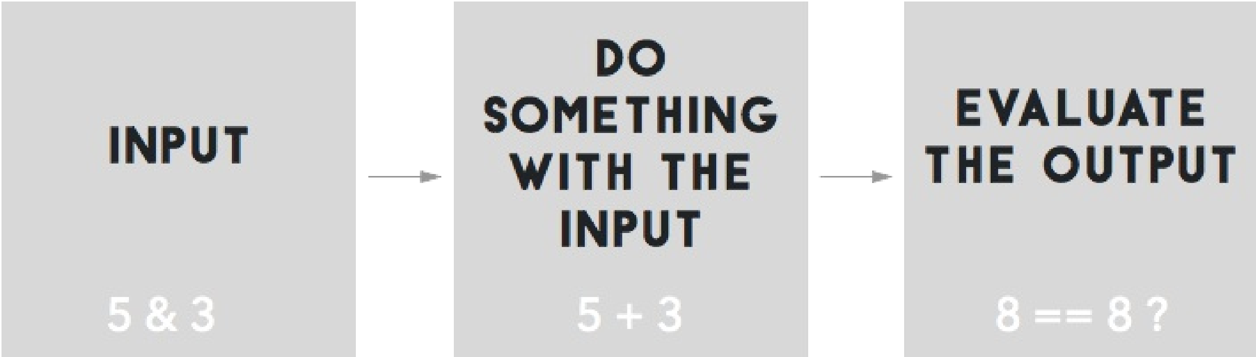
```
override func setUp() {
    super.setUp()

    let storyboard = UIStoryboard(name: "Main", bundle: NSBundle.mainBundle())
    vc = storyboard.instantiateInitialViewController() as! ViewController
}
```

记住，所有测试方法命名必须以关键字test开头，否则Xcode不会识别它。添加一个新的方法testPercentageCalculator()来验证使用ViewController的percentage()方法时是否会正常工作。

```
func testPercentageCalculator() {
}
```

在单元测试中会测试一个代码块是否起了作用。通常来说代码块的测试都是简短并且很典型的几行，用来测试一个特定的方法或函数。在测试时需要提供包含输入值的代码单元，允许使用输入值执行代码，然后看看输出值是否与预期一致。



至于“是否与预期一致”这个问题使用的是XCTAssert函数。最简单的XCTAssert函数是XCTAssert(expression: BooleanType)，它接受一个布尔型表达式(类似5 > 3, 8.90 == 8.90 或true)，若表达式值为true则使测试通过，若为false则测试失败。

试试看! 首先在testPercentageCalculator()方法中添加如下代码。然后将鼠标光标移动到方法名左侧的图标那里，悬停时点击就会开始执行这个测试。

```
func testPercentageCalculator() {
        XCTAssert(true)
}
```

如果一切正常，测试通过后会在方法旁显示一个绿色的标记。

```
func testPercentageCalculator() {
    XCTAssert(true)
}
```

验证百分比计算

该进入正题了: 测试percentage()方法! 作为vc的属性调用这个方法，vc就是ViewController的实例对象。给它两个浮点数，比方说50和50，将结果保存进一个常量p，此时p应该为25(50的50%是25)。用XCTAssert(p == 25)来检验，执行测试方法。使用如下代码替换testPercentageCalculator()方法的内容:

```
func testPercentageCalculator() {
        // Should be 25
        let p = vc.percentage(50, 50)
        XCTAssert(p == 25)
}
```

测试成功了，意味着ViewController的percentage()方法起到了作用，需要去其他地方寻找bug，有可能在updateLabels()方法中？

验证标签

添加一个新的测试方法testLabelValuesShowedProperly()验证标签是否能正确显示文本内容。这次调用ViewController中的updateLabels()方法，检验每个标签是否会显示我们期望的内容。

注意你给了XCTAsset函数一个新的参数:消息字符串. 由于我们要检查多个值(调用3个XCTAssert)所以这么做很方便。如果测试不成功，这条消息会准确地告诉我们哪里有问题。

```
func testLabelValuesShowedProperly() {
        vc.updateLabels(Float(80.0), Float(50.0), Float(40.0))

        // The labels should now display 80, 50 and 40
        XCTAssert(vc.numberLabel.text == "80.0", "numberLabel doesn't show the right t
ext")
        XCTAssert(vc.percentageLabel.text == "50.0%", "percentageLabel doesn't show th
e right text")
        XCTAssert(vc.resultLabel.text == "40.0", "resultLabel doesn't show the right t
ext")
}
```

当执行这个方法时会得到一个错误，三个标签都为空。这怎么可能？

在故事板文件中创建了这些标签，因此在视图加载时它们初始化了，不过在单元测试中loadView()方法并未被触发，并没有创建标签所以为空。对这个问题的解决方法可以是调用一下vc.loadView()，不过Apple在她的文档中不推荐这么做，因为对象在加载后再次加载会引起内存泄漏。

应该访问vc的view属性，这样的话不仅仅是loadView()可以触发所有视图所需要的方法。修改testLabelValuesShowedProperly()中的代码：

```
func testLabelValuesShowedProperly() {
        let _ = vc.view
        vc.updateLabels(Float(80.0), Float(50.0), Float(40.0))

        // The labels should now display 80, 50 and 40
        XCTAssert(vc.numberLabel.text == "80.0", "numberLabel doesn't show the right t
ext")
        XCTAssert(vc.percentageLabel.text == "50.0%", "percentageLabel doesn't show th
e right text")
        XCTAssert(vc.resultLabel.text == "40.0", "resultLabel doesn't show the right t
ext")
}
```

注意这里使用了下划线(_) 作为常量名，这是因为我们不需要view也不会用到它，告诉编译器“假装访问一下view触发它的方法就行了”。

执行测试(如果你想执行测试类中的所有测试可以点击“class PercentageCalculatorTests”旁的方形图标)

```
    func testLabelValuesShowedProperly() {
        let _ = vc.view
        vc.updateLabels(Float(80.0), Float(50.0), Float(40.0))

        // The labels should now display 80, 50 and 40
        XCTAssert(vc.numberLabel.text == "80.0", "numberLabel doesn't show the right text")
        XCTAssert(vc.percentageLabel.text == "50.0", "percentageLabel doesn't    XCTAssertTrue failed - percentageLabel doesn't show the right text
            show the right text")
        XCTAssert(vc.resultLabel.text == "40.0", "resultLabel doesn't show the right text")
    }                                                                        XCTAssertTrue failed - resultLabel doesn't show the right text
}
```

来修复Bug

如你所见，测试失败了! 我们在方法中输入的详细错误信息会帮助我们迅速识别出引起bug的潜在原因。测试结果告诉我们resultsLabel没有显示正确的文本。来进入ViewController看看标签的text属性是如何设置的，在ViewController.swift中的updateLabels()中我们可以发现引起bug的原因:

```
self.resultLabel.text = "\(rV + 10)"
```

应该改为:

```
self.resultLabel.text = "\(rV)"
```

改下你的代码然后再次运行测试，现在就一切正常了!

总结

在这篇教程中，学习了Xcode中的单元测试与如何帮助你在代码中找到bug，单元测试也可以用来进行性能测试与异步测试。另一件你也许感兴趣的就是UI测试，通过记录app的行为来测试你的app在实际的生产环境中的显示情况，可以看看这个WWDC视频来了解一下UI测试。

可以从Github上下载最后的工程

Bond Tutorial: Bindings in Swift

学习Bond：Swift中的数据绑定

> - 原文链接 : Bond Tutorial: Bindings in Swift
> - 原文作者 : Colin Eberhardt
> - 译者 : yrq110

> 注意: 升级到Xcode 7.3, iOS 9.3, Swift 2.2. 2016.04.08

对UI代码的手动绑定感到疲倦吗？我想是的。这是个必要的工作我们不得不去做，不是吗？

Swift Bond是一个绑定框架，可以免去在绑定UI时的繁琐工作，这样就有了更多的时间去思考如何解决应用中的问题。

在这篇Bond教程中，会学着使用Bond来节省时间，写出可维护的代码。Bond与ReactiveCocoa很像，在文章的最后我会说一下，现在让我们来使用Bond进行绑定吧 ... Swift Bond。

> 注意: 对于不知道Swift Bond的读者，这是一个框架的名字,来源于小说中的间谍-James Bond, 他总是这样介绍自己“Bond … James Bond.” ，他很喜欢干马天尼，“摇匀，不要搅拌”。放轻松！准备进入Swift Bond的世界！

目录

- 添加搜索设置
- 绑定日期
- 数据约束

快速绑定

启动Xcode开始写一些代码。

首先，下载开始工程，包含了应用的基本框架。

下载后，首先确保你安装了CocoaPods(如果没有，建议你看看这篇起步教程)。打开终端运行pod install进行安装，像下面这样:

```
$ pod install
Updating local specs repositories
Analyzing dependencies
Downloading dependencies
Installing Bond (4.3.0)
Installing DatePickerCell (1.0.4)
Generating Pods project
Integrating client project
[!] Please close any current Xcode sessions and use `BindingWithBond.xcworkspace` for
this project from now on.
Sending stats
Pod installation complete! There is 1 dependency from the Podfile and 1 total
pod installed.
```

用Xcode打开刚生成的workspace文件，构建并运行，会看到如下图:

一切正常吗? 是时候创建你的第一个绑定了! 打开ViewController.swift在viewDidLoad()中添加如下代码:

```
searchTextField.bnd_text.observe {
  text in
  print(text)
}
```

Bond框架给UIkit控件添加了各种属性，在这里，bnd_text是一个关于UITextField text属性的观测量(observable)，观测量包含了事件流。 别担心，关于这个会详尽地介绍。现在构建并运行一下，输入“hello”看看发生了什么。

观察一下Xcode控制器台中的输出:

```
Optional("")
Optional("h")
Optional("he")
Optional("hel")
Optional("hell")
Optional("hello")
```

每当你键入一个字符时，代码内的闭包表达式都会执行一次，显示应用中text field的当前状态。

这表明了可以监测到例如属性改变等的事件， 你将会看到这其中蕴含着怎样的力量。

数据操作

回到ViewController.swift中，更新你的代码:

```
let uppercase = searchTextField.bnd_text
  .map { $0?.uppercaseString }

uppercase.observe {
    text in
    print(text)
  }
```

如果你用过map来对swift数组进行变换操作的话应该能猜到上述代码做了些什么。

> 注意：如果你对map方法不太熟悉可以看看函数式swift

构建并运行，然后输入hello

```
Optional("")
Optional("H")
Optional("HE")
Optional("HEL")
Optional("HELL")
Optional("HELLO")
```

使用map操作对btn_text的输出进行了变换操作，返回了一个观测量保存进uppercase中。闭包表达式再一次随着事件变化而执行。

执行完map操作后的结果还是一个观测量，因此不需要使用中间变量进行赋值。更新你的代码:

```
searchTextField.bnd_text
  .map { $0?.uppercaseString }
  .observe {
    text in
    print(text)
  }
```

这么写就有了优雅的"流式"语句的风格，构建并运行应用执行相同的操作看看。 你也许觉得这样没啥，印象并不深。让我给你展示一下最棒的代码是怎样的?

使用如下代码替换当前代码:

```
searchTextField.bnd_text
  .map { $0!.characters.count > 0 }
  .bindTo(activityIndicator.bnd_animating)
```

构建并运行你的应用，会注意到活动指示器只会在text field中有文本时才会执行动画!这真是个令人印象深刻的技巧，只用三行代码就实现了，比James Bond的汽车追逐戏更令人印象深刻:]

如你所见，Swift Bond提供了一种流畅并明确的方式去描述应用与UI中的数据流。简化了你应用的代码，更容易理解，并能迅速达成目标。

在下一章你会学到一些工作原理，现在花点时间了解一下它的本事是值得的-Bond … Swift Bond.

工作原理

在做一个更有意义的app之前我们先来看看Bond有哪些功能，我建议你浏览一下Bond的类型(按住cmd点击)，看看注释解释。

在刚才代码的基础上开始编写，bnd_text是一个包含String?属性的Bond观测量，观测量是EventProducer的一个子类。

Event producers会根据由泛型类型定义的数据变化触发相应的事件。

当你调用observe方法时，就将闭包表达式注册为了一个观察者。每次触发事件时，都会用事件数据(由泛型类型定义)调用闭包表达式。

比如，考虑一下一开始写的代码:

```
searchTextField.bnd_text.observe {
    text in
    print(text)
  }
```

调用observe，每当searchTextField的text属性改变时就会执行一次闭包中的代码。

> 注意：这基于一个经典的设计模式 - 观察者模式.

EventProducer遵守EventProducerType协议, 也就是定义map操作时基于的一个协议扩展。你可能听过"面向协议编程"，这就是个很好的例子。

观测量在EventProducer接口上添加了一个值属性，允许它去构建一个属性或变量。通过值属性可以使用'get'或'set'来操作这个值，并能观察它的变化，这得益于EventProducer的功能性。

Bond扩展了UIKit，添加了响应观测量属性。比如UITextField有一个text属性，与之对应的就是观测量bnd_text。所以get或set'bnd_text.value'就等价于对text属性进行相应的操作，使用观测量的话就可以观测到它的改变。

最后一块拼图就是绑定了，bindTo方法连接一对观测量属性，用来同步变化。像下面这样将活动指示器的动画属性与field的text长度绑定:

```
searchTextField.bnd_text
  .map { $0!.characters.count > 0 }
  .bindTo(activityIndicator.bnd_animating)
```

MVVM

Bond使UI属性间的绑定变得很轻松，不过并不是想要的目标。Bond是一个支持模型-视图-视图模型(Model-View-ViewModel,MVVM)模式的框架，如果你没用过这种模式，我建议你看看我以前的教程中的第一部分。 理论知识差不多了，来写写代码吧:]

添加一个视图模型

首先，移除ViewController.swift中的示例代码，一个可信度更高的应用结构应有如下的viewDidLoad()方法:

```
override func viewDidLoad() {
  super.viewDidLoad()
}
```

创建名为PhotoSearchViewModel.swift的文件, 添加如下代码:

```
import Foundation
import Bond

class PhotoSearchViewModel {
  let searchString = Observable<String?>("")

  init() {
    searchString.value = "Bond"

    searchString.observeNew {
      text in
      print(text)
    }
  }
}
```

这里创建了一个简单的包含一个属性的视图模型，使用"Bond"作为值进行初始化。

打开ViewController.swift，在类的顶部添加如下属性:

```
private let viewModel = PhotoSearchViewModel()
```

如下修改viewDidLoad():

```
override func viewDidLoad() {
  super.viewDidLoad()
  bindViewModel()
}
```

在类中添加这个方法:

```
func bindViewModel() {
  viewModel.searchString.bindTo(searchTextField.bnd_text)
}
```

bindViewModel()方法就是之后需要添加绑定的地方，现在仅仅连接text field与视图模型就可以了。

构建并运行应用，会看到显示了文本‘Bond’:

视图模型现在观测的是searchString的变化，不过你会发现在text field中输入时，视图模型并没有及时更新。怎么回事？

现在的绑定是单向的，意味着变化仅仅由源头(视图模型属性)传递给终点(text field的bnd_text属性)

怎么使反向也能传递变化呢? 很简单 — 如下修改绑定方法:

```
viewModel.searchString.bidirectionalBindTo(searchTextField.bnd_text)
```

真轻松! 构建并运行, 输入“Bond Love”:

```
Optional("Bond ")
Optional("Bond L")
Optional("Bond Lo")
Optional("Bond Lov")
Optional("Bond Love")
```

很棒 — 现在确信了text field的变化会传递给视图模型了。

由观测量创建观测量

是时候对观测量映射部分做一些有用的事情了。回到PhotoSearchViewModel.swift，在视图模型中添加如下属性:

```
let validSearchText = Observable<Bool>(false)
```

这个布尔属性标识着当前的searchString是否可用。

依旧在视图模型中添加如下初始化代码:

```
searchString
  .map { $0!.characters.count > 3 }
  .bindTo(validSearchText)
```

在这里将searchString观测量映射到一个检测字符串长度是否大于3个字符的布尔值，接着与validSearchText属性相绑定。

在ViewController.swift中找到bindViewModel方法添加如下代码:

```
viewModel.validSearchText
  .map { $0 ? UIColor.blackColor() : UIColor.redColor() }
  .bindTo(searchTextField.bnd_textColor)
```

这里将validSearchText属性映射到了颜色，然后与text field的文本颜色属性相绑定。

构建并运行:

结果就是，当文本太短不能构成一个可用的搜索关键字时，就会被标注为红色。

Bond将属性间的联系与变换变得非常简单，简单的就像...

派! 不错, 就像这么简单

准备搜索

这个app会根据用户的输入在500px 行检索对应的图片。 giving the same kind of immediate feedback users are used to from Google. 每当searchString视图模型的属性改变时触发一次搜索，不过这可能会引起大量的请求。最好用节流的查询方法，限制每秒最多一次或两次查询。 用Bond很轻松就能实现!

在PhotoSearchViewModel.swift的初始化方法中添加如下代码:

```
searchString
  .filter { $0!.characters.count > 3 }
  .throttle(0.5, queue: Queue.Main)
  .observe {
    [unowned self] text in
    self.executeSearch(text!)
  }
```

这里过滤掉searchString中的不可用值(长度 <= 4), 得到我们需要的值，然后限制变化的速度，最快每0.5秒会接受一个通知。

observe的闭包表达式中调用了executeSearch method, 因此需要在后面添加一下:

```
func executeSearch(text: String) {
  print(text)
}
```

现在这个方法只是打印出结果，可以清楚的看到节流的效果。

构建并运行，试着输入一些文本。

会看到如下的结果，一些经过节流处理的事件，最快每0.5秒显示一条文本

```
Bond
Bond Th
Bond Throttles
Bond Throttles Good!
```

如果没有Bond，要实现这个功能会挺费劲的。

使用500px API

在这个app中需要使用500px API。选择它是因为其具有相关的简易接口与认证机制。

需要先注册成为开发者才能使用接口。非常快捷，易用，而且是免费！ 在这个地址中注册: https://500px.com/signup

注册完成后会跳转到应用界面: https://500px.com/settings/applications. 在这里会看到一个允许你注册应用的界面。

填写注册表单 (有一小部分是必填) ，创建应用。

点击应用并复制一下key，每个请求都是使用这个key来访问500px的。

开始工程中已经包含了所需要的在500px中查询的代码，会在Model group中找到。

回到PhotoSearchViewModel.swift文件,添加如下属性:

```
private let searchService: PhotoSearch = {
  let apiKey = NSBundle.mainBundle().objectForInfoDictionaryKey("apiKey") as! String
  return PhotoSearch(key: apiKey)
}()
```

初始化了PhotoSearch类，这个类提供了一个500px的Swift检索API，通过key来访问。 key应该在plist文件中所包含. 打开Info.plist编辑一下apiKey, 添加500px提供的key:

在PhotoSearchViewModel.swift中如下更新executeSearch方法:

```
func executeSearch(text: String) {
  var query = PhotoQuery()
  query.text = searchString.value ?? ""

  searchService.findPhotos(query) {
    result in
    switch result {
    case .Success(let photos):
      print("500px API returned \(photos.count) photos")
    case .Error:
      print("Sad face :-(")
    }
  }
}
```

构建了一个PhotoQuery对象，包含了查询的参数。 接着调用了searchService的findPhotos方法执行搜索，使用异步回调的方式返回结果，通过枚举值来表现成功或失败的情况。

构建并运行。视图模型会用结构化的方式来初始化查询，会看到如下的输出日志：

```
500px API returned 20 photos
```

如果出错了会输出:

```
Sad face :-(
```

如果出错了，检查下API key与网络连接，祈祷一下，然后再试一次! 如果还不行的话有可能500px的API挂掉了， 不过99%的情况下是你输错了API。仔细检查下输入的key是否与500px中应用的key相同。

渲染结果

如果能看到查询结果返回的那些图像岂不是更有趣？让我们来实现它吧。

在PhotoSearchViewModel.swift中添加如下属性:

```
let searchResults = ObservableArray<Photo>()
```

顾名思义，这时一个观测量的特殊类型，支持数组功能。

在上手之前，我建议看一下ObservableArray的更多细节，按住cmd点击它看看有关的Bond API。

ObservableArray与Observable类似，都属于一种EventProducer，意味着可以设置监测数组改变的事件。此时，事件会触发ObservableArrayEvent实例。

ObservabelArrayEvent基于ObservableArrayOperation枚举对数组中发生的变化进行了编码。

使用间接枚举定义枚举值的关联:

```
public indirect enum ObservableArrayOperation<ElementType> {
  case Insert(elements: [ElementType], fromIndex: Int)
  case Update(elements: [ElementType], fromIndex: Int)
  case Remove(range: Range<Int>)
  case Reset(array: [ElementType])
  case Batch([ObservableArrayOperation<ElementType>])
}
```

如你所见，由观测量数组触发的事件是很详细的，这些事件描述了发生的变化，并没有通知新数组值的观测者。

这种程度的详尽是有充分的理由的。可以使用一个Bond观测量将一个数组绑定到UI，可能是一个列表视图。然而如果绑定到一个单一部件的话，观测量只能表明 某些量 改变了，作为结果整个UI都需要被重构。观测量数组所提供的详细状态提供了更加高效的UI更新方式。

> 注意：ReactiveCocoa没有观测量数组的概念， 这是我之前偶然发现的。你可以在我博客中找到解决方案(很像Bond)。

现在你知道观测量数组是什么了，是时候来使用它了。

在PhotoSearchViewModel.swift中找到executeSearch方法，如下更新case Success中的代码:

```
  case .Success(let photos):
    self.searchResults.removeAll()
    self.searchResults.insertContentsOf(photos, atIndex: 0)
```

这里清空了数组，添加了新的结果。

在ViewController.swift中的bindViewModel里添加如下代码:

```
viewModel.searchResults.lift().bindTo(resultsTable) { indexPath, dataSource, tableView
in
  let cell = tableView.dequeueReusableCellWithIdentifier("MyCell", forIndexPath: index
Path)
                                                                              as! PhotoTableVi
ewCell
  let photo = dataSource[indexPath.section][indexPath.row]
  cell.title.text = photo.title

  let qualityOfServiceClass = QOS_CLASS_BACKGROUND
  let backgroundQueue = dispatch_get_global_queue(qualityOfServiceClass, 0)
  cell.photo.image = nil
  dispatch_async(backgroundQueue) {
    if let imageData = NSData(contentsOfURL: photo.url) {
      dispatch_async(dispatch_get_main_queue()) {
        cell.photo.image = UIImage(data: imageData)
      }
    }
  }
  return cell
}
```

Bond中有一个为绑定观测量数组与列表视图所准备的EventProducerType扩展协议。这个绑定需要输入一个列表视图实例与一个闭包表达式，像上面看到的那样，闭包中的代码很像标准的列表视图datasource设置代码。

再看看其中的细节，首先调用了观测量数组的lift方法。Bond支持以嵌套数组结构为目录的列表视图。lift操作将观测量数组转换成了一个无目录列表所有需要的嵌套表单。

闭包表达式中是标准的列表视图列表项的关联代码，列表出列，设置多个属性。注意，图像的下载是基于一个后台队列，为了保持UI的持续响应。

构建并运行应用:

当在文本框内输入时，UI会自动更新，是不是很酷。

些许UI的天分

是时候对UI做的什么了！

在PhotoSearchViewModel.swift中添加如下属性y:

```
let searchInProgress = Observable<Bool>(false)
```

更新executeSearch方法中的代码t:

```
func executeSearch(text: String) {
  var query = PhotoQuery()
  query.text = searchString.value ?? ""

  searchInProgress.value = true

  searchService.findPhotos(query) {
    [unowned self] result in
    self.searchInProgress.value = false
    switch result {
    case .Success(let photos):
      self.searchResults.removeAll()
      self.searchResults.insertContentsOf(photos, atIndex: 0)
    case .Error:
      print("Sad face :-(")
    }
  }
}
```

这里在进行查询操作之前将属性直接设为了true，当结果异步返回时则设为false。

在ViewController.swift中，给bindViewModel添加如下代码:

```
viewModel.searchInProgress
  .map { !$0 }
  .bindTo(activityIndicator.bnd_hidden)

viewModel.searchInProgress
  .map { $0 ? CGFloat(0.5) : CGFloat(1.0) }
  .bindTo(resultsTable.bnd_alpha)
```

在查询进行时会显示一个正在加载的指示器，并且会降低列表视图的透明度。

构建并运行一下，看看效果:

现在应该体会到使用Bond的好处了吧！ 效果像马提尼一样美妙 :]

处理错误

如果500px的查询失败了，ap仅仅会输出错误信息到控制台中。这里应该以一种有参考意义并规范的方式将失败信息反馈给用户。

问题来了，怎么定义这个模型？一个错误不像视图模型属性是状态的改变，而是瞬态发生的事件。

答案很简单: 一个EventProducer，而不是观测量。在PhotoSearchViewModel.swift中添加如下属性:

```
let errorMessages = EventProducer<String>()
```

接着，如下修改executeSearch中Error情况下的代码:

```
  case .Error:
    self.errorMessages
      .next("There was an API request issue of some sort. Go ahead, hit me with that 1
-star review!")
```

在ViewController.swift中修改bindViewModel:

```
viewModel.errorMessages.observe {
  [unowned self] error in

  let alertController = UIAlertController(title: "Something went wrong :-(",
                                                        message: error, preferredStyle
: .Alert)
  self.presentViewController(alertController, animated: true, completion: nil)
  let actionOk = UIAlertAction(title: "OK", style: .Default,
    handler: { action in alertController.dismissViewControllerAnimated(true, completio
n: nil) })

  alertController.addAction(actionOk)
}
```

这里设置了errorMessages属性触发的事件，使用UIAlert来显示错误信息。

构建并运行应用，然后断开网络连接或者移除key来看看显示的错误信息:

完美 :]

添加搜索设置

当前应用的查询是基于500px API的简单搜索功能，如果看看API文档的话会发现它支持很多其他的参数。

如果点击app中的Settings按钮，会看到已经有了一个简单的搜索设置UI，现在需要来实现它了。

在视图模型group中添加一个名为PhotoSearchMetadataViewModel.swift的文件，输入如下内容:

```
import Foundation
import Bond

class PhotoSearchMetadataViewModel {
  let creativeCommons = Observable<Bool>(false)
  let dateFilter = Observable<Bool>(false)
  let minUploadDate = Observable<NSDate>(NSDate())
  let maxUploadDate = Observable<NSDate>(NSDate())
}
```

在PhotoSearchViewModel.swift中添加如下属性:

```
let searchMetadataViewModel = PhotoSearchMetadataViewModel()
```

修改executeSearch方法，在设置query.text属性的那行后面添加:

```
query.creativeCommonsLicence = searchMetadataViewModel.creativeCommons.value
query.dateFilter = searchMetadataViewModel.dateFilter.value
query.minDate = searchMetadataViewModel.minUploadDate.value
query.maxDate = searchMetadataViewModel.maxUploadDate.value
```

上面的代码将视图模型的状态拷贝给了PhotoQuery对象。

这个视图模型将会与settings视图控制器绑定，是时候来实现了，打开SettingsViewController.swift添加如下属性:

```
var viewModel: PhotoSearchMetadataViewModel?
```

那么如何设置这个属性? 当Settings按钮被按下时，故事板会执行一个segue，可以“拦截”这个进程，给视图控制器输入一些数据。

在ViewController.swift中添加如下方法:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  if segue.identifier == "ShowSettings" {
    let navVC = segue.destinationViewController as! UINavigationController
    let settingsVC = navVC.topViewController as! SettingsViewController
    settingsVC.viewModel = viewModel.searchMetadataViewModel
  }
}
```

这里保证了在ShowSettings segue执行时，目标视图控制器正确加载了视图模型。

在SettingsViewController.swift中viewDidLoad()方法的尾部添加如下代码:

```
bindViewModel()
```

然后添加这个函数的实现:

```
func bindViewModel() {
  guard let viewModel = viewModel else {
    return
  }
  viewModel.creativeCommons.bidirectionalBindTo(creativeCommonsSwitch.bnd_on)
}
```

上面的代码绑定了视图模型属性与响应的switch开关。

构建并运行应用，打开Settings并触发开关。如果打开的话，会发现加载时返回的照片会有所不同。

绑定日期

Settings界面还有一些需要实现的功能，这就是接下来的任务。

在SettingsViewController.swift中的bindViewModel()中添加如下代码:

```
viewModel.dateFilter.bidirectionalBindTo(filterDatesSwitch.bnd_on)

let opacity = viewModel.dateFilter.map { $0 ? CGFloat(1.0) : CGFloat(0.5) }
opacity.bindTo(minPickerCell.leftLabel.bnd_alpha)
opacity.bindTo(maxPickerCell.leftLabel.bnd_alpha)
opacity.bindTo(minPickerCell.rightLabel.bnd_alpha)
opacity.bindTo(maxPickerCell.rightLabel.bnd_alpha)
```

将dateFilter开关与视图模型绑定在了一起，并减少了闲置的datePicker单元格的数量。

datePicker单元格包含一个datePicker与几个标签，使用CocoaPod中的DatePickerCell实现，这里有个小问题：Bond并不支持绑定这种单元格类型所暴露的属性。

如果你看看DatePickerCell API的话，会发现一个date属性包含标签与picker。可以对picker进行双向绑定，不过这样的话就意味着跳过了设置标签内容的逻辑。

幸运的是可以“手动”的进行双向绑定，同时观测model与datePicker，很直接的方法。

将如下方法添加到SettingsViewController.swift中:

```
private func bind(modelDate: Observable<NSDate>, picker: DatePickerCell) {
  modelDate.observe {
    event in
    picker.date = event
  }

  picker.datePicker.bnd_date.observe {
    event in
    modelDate.value = event
  }
}
```

然后将这两行添加到bindViewModel()中:

```
bind(viewModel.minUploadDate, picker: minPickerCell)
bind(viewModel.maxUploadDate, picker: maxPickerCell)
```

构建并运行，欢呼吧! 日期现在被正确绑定了:

数据约束

现在，用户创建的日期过滤器可能不太符合逻辑，比如，最早日期比早晚日期要晚。

实现对其的约束的确挺容易的。在PhotoSearchMetadataViewModel.swift中添加如下初始化方法:

```
init() {
  maxUploadDate.observe {
    [unowned self]
    maxDate in
    if maxDate.timeIntervalSinceDate(self.minUploadDate.value) < 0 {
      self.minUploadDate.value = maxDate
    }
  }
  minUploadDate.observe {
    [unowned self]
    minDate in
    if minDate.timeIntervalSinceDate(self.maxUploadDate.value) > 0 {
      self.maxUploadDate.value = minDate
    }
  }
}
```

上面对每个日期都添加了observe，如果出现了最小>最大的情况，则值会根据情况发生改变。

构建并运行一下，看看效果。故意设置最晚日期比最早日期要早，它会自动将其调整为最早日期。

也许你会发现有一个违和的地方：当改变搜索设置时，app不会重复进行查询，这也许会困扰你的用户。理论上来说，app在设置改变时应该再次执行搜索。

每个设置都是一个观测量属性，应该观测它们全部，然而这样需要大量的重复性代码。幸运地是这里有一种更好的方式!

在PhotoSearchViewModel.swift中init()的底部添加如下代码:

```
combineLatest(searchMetadataViewModel.dateFilter, searchMetadataViewModel.maxUploadDat
e,
  searchMetadataViewModel.minUploadDate, searchMetadataViewModel.creativeCommons)
  .throttle(0.5, queue: Queue.Main)
  .observe {
    [unowned self] _ in
    self.executeSearch(self.searchString.value!)
  }
```

combineLatest函数整合了所有观测量，允许你将它们当做一个量来看待。上面的代码执行了整合，限流与执行查询的操作。

构建并运行一下，看看效果。尝试改变日期，或拨动开关。每当你改变一些东西时就会更新新的搜索结果! 是不是很简单?! :]

What’s New in Swift 3

Swift 3中的新特性

> - 原文链接：What’s New in Swift 3
> - 原文作者：COSMIN PUPĂZĂ
> - 译者：yrq110

苹果在Xcode 8 beta版中集成了Swift 3，并且将在今年年末发布最终版本。这是3的第一个版本，开源并且可运行于MacOS与Linux。如果你从去年12月开始就跟进Swift-Evolution的话，甚至包括已经在IBM沙盒中体验过的boy，应该知道这里面有很多的变化。如果现在在Xcode8中编译你的app的话，肯定会崩溃的。

Swift 3中的变化可以被分为两类:

- 移除Swift 2.2中已经被弃用的功能
- 语言的现代化问题

从被移除的条目开始，这个比较容易理解，在Xcode7.3中使用它们应该已经出现警告了。

目录

++ 与 -- 运算符

继承自C语言的自增与自减的操作符与它的功能是显而易见的，对一个量进行加1或减1:

```
var i = 0
i++
++i
i--
--i
```

然而，当遇到该选择哪个值进行操作时(运算前和运算后)事情会变复杂。每种操作符都有两种风格：前缀和后缀，它们都是内部函数，得益于操作符重载它会返回一个你可能会使用也可能丢弃的值。

这对初学者来说很不友好，因此需要被移除，转而使用加法(+=)与减法(-=)赋值运算符:

```
var i = 0
i += 1
i -= 1
```

当然，你也可以使用常规的加法(+)与减法(-)运算符，虽然使用复合赋值运算符的话更加简便:

```
i = i + 1
i = i - 1
```

> 延伸阅读: 如果你想了解更多关于这个变化的信息，可以看看 Chris Lattner关于移除++与--操作符的建议。

C语言风格的for循环已成为历史

最常用的增减量运算符就是C语言风格的for循环，这个运算符的移除意味着这种特性也将消失。

如果你有一些编程经历的话，对打印数字1到10这个问题，也许会写出这样的for循环语句:

```
for (i = 1; i <= 10; i++) {
  print(i)
}
```

在Swift 3中将不再允许这样做。这是Swift 3中对应的写法 - 注意闭值区间运算符(...)的使用:

```
for i in 1...10 {
  print(i)
}
```

你也可以使用带有一个闭包的for-each循环来实现 - 在这里查看更多关于循环的信息

```
(1...10).forEach {
  print($0)
}
```

> 延伸阅读: 如果你想了解更多关于这个变化的信息，可以看看Erica Sadun关于移除C风格for循环的建议。

将var从函数参数中移除

通常函数的输入参数都被定义为常量，即在函数体中你不能修改它们，然而某些情况下需要将它们声明为变量才能派上用场。在Swift 2中，你可以使用var关键字将一个函数参数标记为变量，一旦参数被标记为变量，就会在本地创建一个这个值的副本，这样你就可以在函数体内修改它。

举个栗子，下面这个函数会计算出两个数字的最大公约数 - 如果你把数学知识全还给老师了，请读读这个

```
func gcd(var a: Int, var b: Int) -> Int {

  if (a == b) {
    return a
  }

  repeat {
    if (a > b) {
      a = a - b
    } else {
      b = b - a
    }
  } while (a != b)

  return a
}
```

算法很简单: 如果两个数字相等，则返回它们中的一个；否则比较它们，用较大的数减去较小的数，将结果赋值给较大的那一个变量，反复进行直到它们相等，返回其中的一个。如你所见，将a与b标记为变量后即可在函数中改变它们的值。

Swift 3中将不再允许开发者将函数参数设置为变量，因为这也许会使开发者在var与inout(PS:另一个关键字，自行google)间产生困扰。下一个Swift的版本会将var从函数参数中移除。

因此，若要在Swift 3中写出同样的gcd函数，需要不同的途径来使用。你需要将函数参数保存进本地变量中再进行运算:

```
func gcd(a: Int, b: Int) -> Int {

  if (a == b) {
    return a
  }

  var c = a
  var d = b

  repeat {
    if (c > d) {
      c = c - d
    } else {
      d = d - c
    }
  } while (c != d)

  return c
}
```

若你想了解这项移除的相关信息，可以查看这个建议。

一致性的函数参数声明方式

函数的参数列表实际上是元组，因此可以通过输入一个元组来调用函数，只要元组的结构与函数参数的结构匹配即可。拿gcd()函数为例，你可以这样调用它:

```
gcd(8, b: 12)
```

也可以这么调用:

```
let number = (8, b: 12)
gcd(number)
```

如你所见，在Swift 2无需指明第一个参数的标签，不过，第二个以后的参数标签都需要被指明。

这个语句会困扰初学者，因此设计成了规范的标签行为。在Swift 3中，需要这样调用函数:

```
gcd(a: 8, b: 12)
```

你需要明确的表明第一个参数的标签，如果不这么做，Xcode 8就会报错。

对于这个变化你的第一个反应可能是“卧槽！现在的代码中要改好多地方啊”。对的，有大量的变动。对于这个苹果提供了一种方法来解决这个问题。你可以在第一个参数的标签前添加一个下划线:

```
func gcd(_ a: Int, b: Int) -> Int {
...
}
```

这么做的话，就可以像以前那样调用函数了 - 无需指明第一个标签。这会使你在做Swift 2 到 Swift 3的代码移植时会轻松些。

> 延伸阅读: 若你想了解这项变化的相关信息，可以查看这个建议。

选择器字符串将失效

来创建一个按钮，在你点击它时做一些事情。只使用playground，不允许用Interface Builder:

```
// 1
import UIKit
import XCPlayground

// 2
class Responder: NSObject {

  func tap() {
    print("Button pressed")
  }
}
let responder = Responder()

// 3
let button = UIButton(type: .System)
button.setTitle("Button", forState: .Normal)
button.addTarget(responder, action: "tap", forControlEvents: .TouchUpInside)
button.sizeToFit()
button.center = CGPoint(x: 50, y: 25)

// 4
let frame = CGRect(x: 0, y: 0, width: 100, height: 50)
let view = UIView(frame: frame)
view.addSubview(button)
XCPlaygroundPage.currentPage.liveView = view
```

让我们来逐步分析下:

1. 导入UIKit与XCPlayground框架 - 你需要它们创建按钮然后在playground的辅助编辑器(assistant editor)中显示它。

   > 注意: 在Xcode中这样启用辅助编辑器: View -> Assistant Editor -> Show Assistant Editor.

2. 定义一个当用户按下按钮时执行的tap方法，创建一个基于NSObject的按钮响应对象 - 为何是NSObject？因为选择器只适用于OC的方法。

3. 声明button按钮并设置属性。

4. 声明view视图与响应框体，在上面添加button，并且在playground的辅助编辑器中显示它。

注意第3部分的第3行代码。button的选择器是一个字符串。如果你敲错了这个字符串，代码会编译成功不过会在runtime时崩溃，因为没有找到任何响应方法。

为了解决这个在编译时的潜在问题，Swift 3使用#selector()来代替字符串选择器。这使得编译器可以在没有得到正确的方法名时提早检测出问题。

```
button.addTarget(responder, action: #selector(Responder.tap), for: .touchUpInside)
```

> 延伸阅读: 若你想了解这项变化的相关信息，可以查看Doug Gregor的建议。

以上这些是被移除的特性。现在来看看语言的现代化亮点吧。

Key-paths字符串

这个特性有点像前面那个，不过这个是应用于键值编码(KVC)与键值观察(KVO)的:

```
class Person: NSObject {
  var name: String = ""

  init(name: String) {
    self.name = name
  }
}
let me = Person(name: "Cosmin")
me.valueForKeyPath("name")
```

创建了一个符合KVC规范的Person类，使用类指定的初始化值来产生我的身份，使用对应的key-path来决定我的名字。检查一下，如果你输错了我会生气的！

幸运的是，在Swift 3中不会发生这种事，Key-path字符串已经被#keyPath()表达式所替代:

```
class Person: NSObject {
  var name: String = ""

  init(name: String) {
    self.name = name
  }
}
let me = Person(name: "Cosmin")
me.value(forKeyPath: #keyPath(Person.name))
```

> 延伸阅读: 若你想了解这项变化的相关信息，可以查看David Hart的建议。

取消Foundation类型的NS前缀

JSON解析是一个典型的例子:

```
let file = NSBundle.mainBundle().pathForResource("tutorials", ofType: "json")
let url = NSURL(fileURLWithPath: file!)
let data = NSData(contentsOfURL: url)
let json = try! NSJSONSerialization.JSONObjectWithData(data!, options: [])
print(json)
```

使用Foundation类去连接文件并提取JSON数据，通过一种合适的途径: NSBundle -> NSURL -> NSData -> NSJSONSerialization.

Swift 3中取消的了NS前缀, 因此这条路径变成了: Bundle -> URL -> Data -> JSONSerialization:

```
let file = Bundle.main().pathForResource("tutorials", ofType: "json")
let url = URL(fileURLWithPath: file!)
let data = try! Data(contentsOf: url)
let json = try! JSONSerialization.jsonObject(with: data)
print(json)
```

> 延伸阅读: 关于这个命名规范的变化，可以查看Tony Parker与Philippe Hausler的建议。

M_PI vs .pi

来根据半径计算一下圆的周长与面积:

```
let r =  3.0
let circumference = 2 * M_PI * r
let area = M_PI * r * r
```

之前的Swift版本中，将M_PI作为pi常量使用。Swift 3将pi整合进了Float, Double与CGFloat类型中:

```
Float.pi
Double.pi
CGFloat.pi
```

上面的代码在Swift 3中会这么写:

```
let r = 3.0
let circumference = 2 * Double.pi * r
let area = Double.pi * r * r
```

由于有类型关联，可以省略掉类型，简便写法:

```
let r = 3.0
let circumference = 2 * .pi * r
let area = .pi * r * r
```

GCD多线程

Grand Central Dispatch(GCD)用于在不阻塞主线程中用户界面情况下进行网络操作。它的底层使用C语言编写，其API的使用对初学者来说有些难度，即使是一些简单的任务，例如创建一个异步队列然后做一些事情:

```
let queue = dispatch_queue_create("Swift 2.2", nil)
dispatch_async(queue) {
  print("Swift 2.2 queue")
}
```

Swift 3中去除了所有在面向对象编程的过程中的模板代码与冗余部分:

```
let queue = DispatchQueue(label: "Swift 3")
queue.async {
  print("Swift 3 queue")
}
```

> 延伸阅读：关于这个变化，可以查看Matt Wright的建议。

核心图形库(Core Graphics)现在更加Swifty

Core Graphics是一个强大的绘图框架，不过它与GCD类似，使用C语言风格的API：

```
let frame = CGRect(x: 0, y: 0, width: 100, height: 50)

class View: UIView {

  override func drawRect(rect: CGRect) {
    let context = UIGraphicsGetCurrentContext()
    let blue = UIColor.blueColor().CGColor
    CGContextSetFillColorWithColor(context, blue)
    let red = UIColor.redColor().CGColor
    CGContextSetStrokeColorWithColor(context, red)
    CGContextSetLineWidth(context, 10)
    CGContextAddRect(context, frame)
    CGContextDrawPath(context, .FillStroke)
  }
}
let aView = View(frame: frame)
```

创建了view框体，扩展了UIView类，重写了drawRect()方法进行自定义绘制，将新的content赋给了view。

Swift 3中通过一个完全不同的途径进行，首先解绑当前的图形上下文，然后执行所有与之相关的绘制操作：

```
let frame = CGRect(x: 0, y: 0, width: 100, height: 50)

class View: UIView {

  override func draw(_ rect: CGRect) {
    guard let context = UIGraphicsGetCurrentContext() else {
      return
    }

    let blue = UIColor.blue().cgColor
    context.setFillColor(blue)
    let red = UIColor.red().cgColor
    context.setStrokeColor(red)
    context.setLineWidth(10)
    context.addRect(frame)
    context.drawPath(using: .fillStroke)
  }
}
let aView = View(frame: frame)
```

> 注意: 在view调用它的drawRect()方法之前上下文是空的，因此可以使用guard来进行解绑 – 在这里了解更多信息。

动词 vs 名词 命名规范

终于有个跟语法有关的了! Swift 3的group方法分为两类: 返回一个值的方法 – 认为是名词，与执行一个类型的行为的方法 – 认为它们是动词。

这里进行了打印数字从10到1的操作:

```
for i in (1...10).reverse() {
  print(i)
}
```

使用reverse()方法对range进行反转。Swift 3将这个操作看做一个名词，返回一个执行了反转指令的range。需要在方法中添加“ed”后缀:

```
for i in (1...10).reversed() {
  print(i)
}
```

最常用的元组用法就是打印一个数组的内容:

```
var array = [1, 5, 3, 2, 4]
for (index, value) in array.enumerate() {
  print("\(index + 1) \(value)")
}
```

Swift 3将这个方法看做一个名词，因为它返回一个包含数组当前索引与值的元组，因此在方法名后添加“ed”:

```
var array = [1, 5, 3, 2, 4]
for (index, value) in array.enumerated() {
  print("\(index + 1) \(value)")
}
```

还有个例子就是数组排序，这里进行了数组升序的排序操作:

```
var array = [1, 5, 3, 2, 4]
let sortedArray = array.sort()
print(sortedArray)
```

Swift 3将这个方法看做一个名词，因为它返回一个排序好的数组。sort方法现在叫做sorted:

```
var array = [1, 5, 3, 2, 4]
let sortedArray = array.sorted()
print(sortedArray)
```

不使用中间变量来进行数组的就地排序。 在Swift 2中需要这么写:

```
var array = [1, 5, 3, 2, 4]
array.sortInPlace()
print(array)
```

使用sortInPlace()对一个可变数组进行排序。Swift 3将这个方法看做一个动词，因为它在执行排序时没有返回任何东西。 它进行的操作仅仅是基础单词所描述的行为，因此 sortInPlace() 现在被命名为sort():

```
var array = [1, 5, 3, 2, 4]
array.sort()
print(array)
```

> 延伸阅读: 对于命名规范的细节，可以查看API的设计指南.

Swift风格的API

Swift 3的API运用了简洁的哲♂学理念 – 省略了无用的单词, 因此如果有冗余或可由上下文关联得到的单词就被去掉了:

- `XCPlaygroundPage.currentPage` -> `PlaygroundPage.current`
- `button.setTitle(forState)` -> `button.setTitle(for)`
- `button.addTarget(action, forControlEvents)` -> `button.addTarget(action, for)`
- `NSBundle.mainBundle()` -> `Bundle.main()`
- `NSData(contentsOfURL)` -> `URL(contentsOf)`
- `NSJSONSerialization.JSONObjectWithData()` -> `JSONSerialization.jsonObject(with)`
- `UIColor.blueColor()` -> `UIColor.blue()`
- `UIColor.redColor()` -> `UIColor.red()`

枚举

Swift 3将枚举的cases看做属性，因此使用小写开头的驼峰式命名，而不是大写开头的驼峰式来标识:

- `.System` -> `.system`
- `.TouchUpInside` -> `.touchUpInside`
- `.FillStroke` -> `.fillStroke`
- `.CGColor` -> `.cgColor`

@discardableResult

在Swift 3中，如果你没有使用函数或方法返回值的话Xcode会显示一个警告，举个栗子:

```
override func viewDidLoad() {
    super.viewDidLoad()

    printMessage(message: "Hello Swift 3!")          Result of call to 'printMessage(message:)' is unused
}

func printMessage(message: String) -> String {
    let outputMessage = "Output : \(message)"
    print(outputMessage)

    return outputMessage
}
```

上面代码中的printMessage方法返回了结果信息，返回值没有被使用。这是个潜在问题，因此Swift 3中的编译器会给出一个警告。

这种情况下没必要使用返回值，你可以在方法声明前添加@discardableResult来避免警告:

```
override func viewDidLoad() {
    super.viewDidLoad()

    printMessage(message: "Hello Swift 3!")
}

@discardableResult
func printMessage(message: String) -> String {
    let outputMessage = "Output : \(message)"
    print(outputMessage)

    return outputMessage
}
```

总结

这就是关于Swift 3的全部内容了。新版的Swift是一个重要的发行版，会使这个语言变得更好，它包含了大量的功能上的变化，对你的现有Swift代码有很大的影响。我希望这篇教程可以帮助你更好的理解这些变化，并且我希望这可以节省一些你移植Siwft项目的时间。

这篇教程中的所有代码都可以在这个Playground工程中找到，我在Xcode8 beta中测试成功了，确保你使用的是Xcode 8然后再运行这个项目。

Building a Chat App in Swift Using Multipeer Connectivity Framework

使用Multipeer Connectivity框架构建一个聊天app

> - 原文链接：Building a Chat App in Swift Using Multipeer Connectivity Framework
> - 原文作者：GABRIEL THEODOROPOULOS
> - 译者：yrq110

`Multipeer Connectivity` - `多点连接`

在iOS编程中，总有一些SDK相比其它而言更有意思更吸引开发者的注意，Multipeer Connectivity就是其中一个。如你所知，MPC框架并不是iOS8中的新东西， 在7中就出现了。之前我写过一些相关的教程，不过说实话，我很惊讶大家对它如此的感兴趣。现在过了一段时间后，又写了一篇新的教程，相信这其中还 有一些东西需要说明下。

你也许会疑惑为何会说这个有些古老的话题，而不是iOS8中的新特性。好吧，我这么做有三个原因：

1. 很多读者都通过邮件告诉我想知道在multipeer connectivity框架中如何处理多种不同的任务，在回答这些问题时，我发现了一些以前没注意到的问题，而且有些是不容易发现的。
2. 在之前的教程中我使用了一个默认的、创建好的视图控制器来实现用户的邀请与建立连接。读者反馈想要看看手动的实现，这也是接下来要做的。
3. 我觉得使用Swift来实现MPC是很有用的，对初学者也很有帮助。



自multipeer connectivity诞生以来，为大量开发者提供了很多实现新点子的可能性。使用一种简单的方式去进行设备间的连接是很诱人的，这也是为何开发者会乐于将它集成在app中。不过如果你还没用过MPC框架的话，我必须提醒你：它有时不是你所期望的那么稳定、健壮，我在我的项目中发现过类似问题，有些开发者也告知过我同样的问题。MPC使用蓝牙和WiFi来连接附近的设备，虽然这听起来很棒很有前景，但是有时由于通信过程中的问题，连接也会失败或变得很慢。若你要传输重要数据的话，这种情况是需要重点考虑的。我建议你使用一个备用的通信解决方案(像一个web服务)，确保有一个备选途径，当MPC失效时app也能继续工作。尽管我这么说，我依然相信MPC是一个对所有iOS开发者都很不错的工具，值得俺再写一篇教程。

我不打算深入MPC的细节中。如果想尝尝鲜的话，看看这篇教程。否则的话，看看我总结的这个MPC的概述。

MPC中包含4个重要的类，分别对应4个概念:

1. Peer-节点 (MCPeerID): 一个节点实则就是一台设备，这通常是需要优先设置的，因为它会作为下一个类实例初始化时的一个参数使用。它包含一个重要的属性 - displayName，这是一台设备对于附近的节点所显示的名字。
2. Session-会话 (MCSession class): 这是两个节点间建立的连接，前提是第一个节点邀请了第二个，并且第二个接受了邀请。一个会话只与两台设备有关。一个第三方设备不能连接进一个已存在的会话。
3. Browser-搜索器 (MCNearbyServiceBrowser class): 这个类的方法被用来寻找附近的设备并邀请他们加入一个会话。作为前提条件，其他设备必须先广播他们自己。在这篇教程中用使用这个类去手动邀请其他设备。
4. Advertiser-广播器 (MCNearbyServiceAdvertiser): 这个类负责广播一个设备，控制这个设备对其他设备可见或不可见，接受或拒绝其他节点的会话邀请。

MPC的逻辑是很简单的: 一台设备(一个节点)使用它的搜索器，寻找周围的其他设备。一台设备必须广播它自己，否则不会被发现。一旦它在搜索时发现了一个或多个节点，就会向其发送一个会话连接的邀请。这个邀请可以在发现周围节点时自动发送，也可以用户手动决定是否发送，这完全取决于应用中国是如何实现的。无论任何情况，只要邀请被接受，会话就会建立，两个节点间就可以收发数据、资源与流。

本文中我的目标是给你演示如果编程实现搜索、邀请与连接其它节点。记住，之后看到的只是其中一种代码实现的方式，很显然，你可以通过使用MPC中提供的所有工具，以你自己想要的方式去实现适合你app的功能。下面的内容只是一个MPC框架实现的例子，实现了精简的从头到尾的编程过程，没有使用任何额外的SDK。我希望在这篇教程最后你们能找到一些所寻找的答案。

最后，如果你从未使用过MPC框架，我建议你扫一眼苹果的官方文档与我之前的教程。哦了不浪费时间了，开始编程吧。

目录

关于Demo App

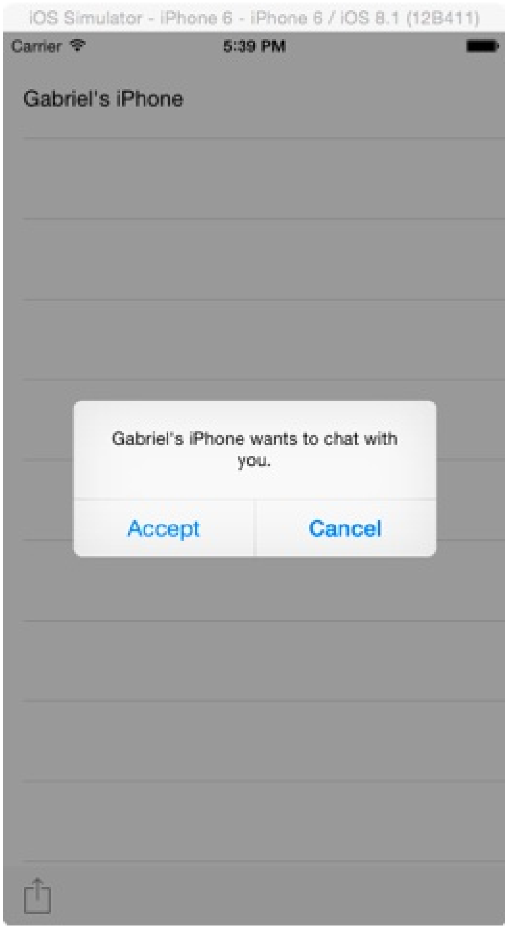
来稍微说下demo应用，一个聊天app。好吧这与之前MPC教程中的聊天app有很大的共同之处，不过让我解释一下:不管我怎么想，都找不到一个比这个更合适的类型，聊天app一直会是首选，我相信在教程的最后你会认同我的看法的。

开始工程在这里下载，将在这个的基础上进行编程，下好后先大致浏览下，以便能快速的上手。

如我所讲，将构建一个聊天app，这个app分为两个视图控制器，第二个会呈现给用户。如果你看看工程中的Interface Builder会发现第一个视图控制器中有一个列表视图，在这个列表视图中将要显示的是MPC框架所搜索到的所有附近的设备，列表中的节点是动态刷新的，消失的已存在节点与新发现的节点都会动态显示在列表中。这里还有一个包含一个按钮的工具栏，后面会使用这个按钮去打开和关闭设备的可发现性，针对MPC中广播的特性所设计。

在点击一个列表项节点时，会询问另一端的用户是否开始聊天。如果他拒绝则什么都不会发生，如果接受的话会在屏幕中显示第二个视图控制器，名为ChatViewController，那么两台设备就可以交换文本信息了。这里包含一个用来撰写信息的textfield和一个显示所有信息的列表视图。在顶部工具栏的按钮用来终止聊天。差不多就是所有细节了，来看看具体的实现吧。

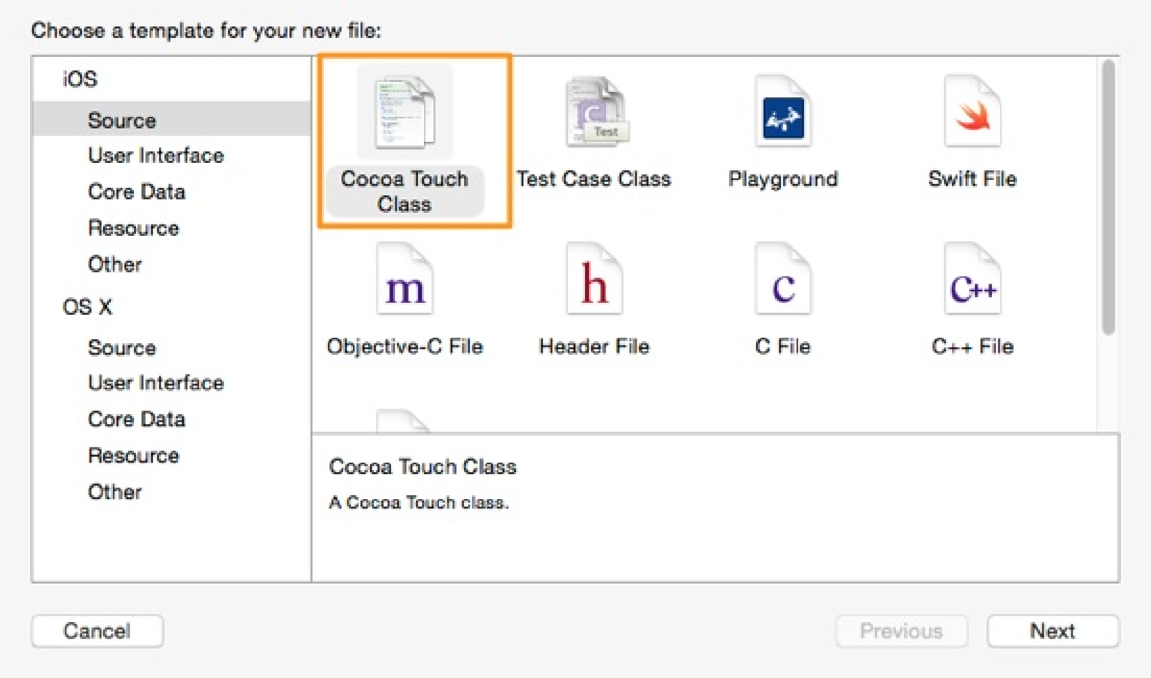
在看app的最终效果图之前，我还要说一件事。尽管我们可以用MPC连接7个甚至更多的设备，在这里只连接一个节点进行多次测试，因为使用会话来连接节点与发现和列出所有附近节点是两码事。在下面的部分你会看到逐一实现的步骤，现在先感受一下吧:



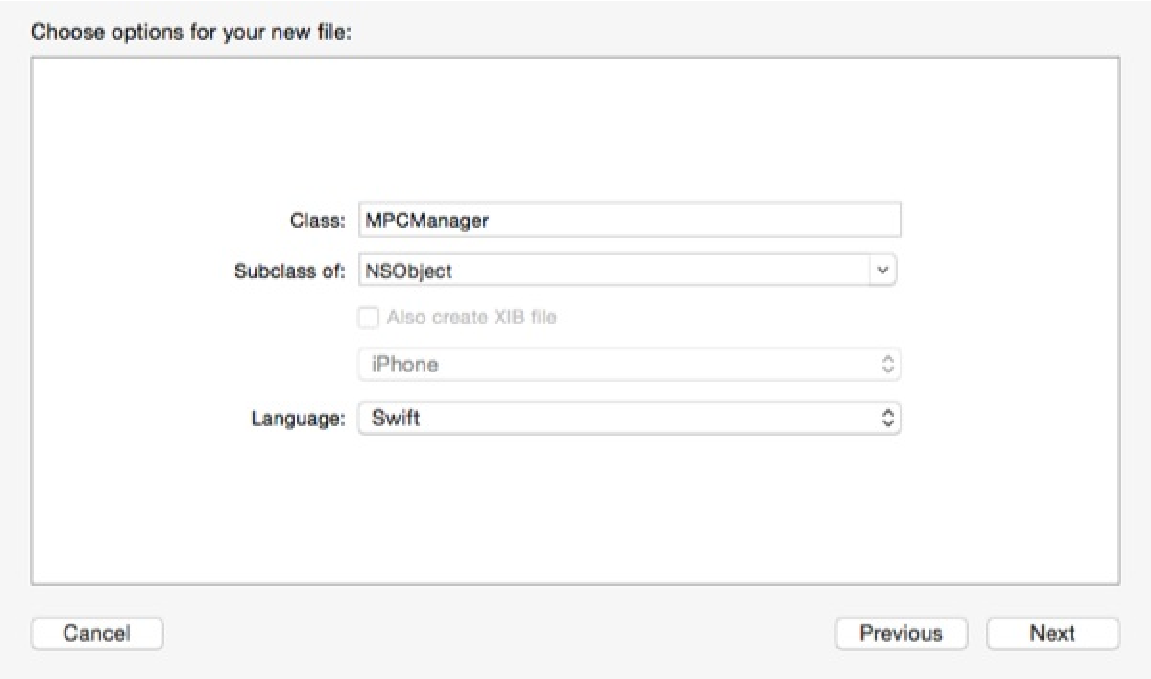
一个自定义类

知道这个工程是做什么的后让我们来创建一个自定义类，在这个类中会处理所有需要实现的MPC框架方法，在最后我们会创建一个使用代理模式的新协议。

当务之急，在Xcode中按住Cmd与N组合键，在出现的引导视图中，选择创建一个新的Cocoa Touch Class:



第二步中，使新的类继承自NSObject类，命名为MPCManager



跟着向导走，确保你正在操作的是MPCManager.swift文件。

在代码的第一行，需要导入multipeer connectivity框架，因此在文件头部添加如下代码:

```
import MultipeerConnectivity
```

接着来声明需要用到的MPC框架中的类的对象，在类中的顶部添加如下行:

```
var session: MCSession!

var peer: MCPeerID!

var browser: MCNearbyServiceBrowser!

var advertiser: MCNearbyServiceAdvertiser!
```

除了上面那些，还需要声明两个以后要用到的变量:

```
var foundPeers = [MCPeerID]()

var invitationHandler: ((Bool, MCSession!)->Void)!
```

在foundPeers数组中会存放所有发现的节点。注意此时与这些节点间不会构建任何连接，只需要知道周围有这些节点。这个数组在声明的同时进行了初始化，因此不需要额外设置为nil，期望数组在发现新对象时已经准备就绪。

invitationHandler是一个完整声明的处理程序，不过现在不会管它，等用的时候再说。

接下来需要使自定义类满足特定的MPC协议。这些协议的委托方法使我们可以去处理multipeer connectivity的相关操作，比如搜索、广播、会话等。如下修改类的第一行，添加协议:

```
class MPCManager: NSObject, MCSessionDelegate, MCNearbyServiceBrowserDelegate, MCNearb
yServiceAdvertiserDelegate
```

我觉得没必要解释每个协议的用途了。

现在来创建一个初始化器来初始化所有的MPC对象。一个一个来，从peer对象开始，它代表了，在初始化时需要提供一个显示的名称(display name)，这个显示名称将会对其他节点可见，并且可以设置为任意字符串。为了使事情简单点，我直接将设备名称作为显示名称，不过我不建议在一个实际的app中这样做，也许你应该让用户自己来输入想要的名字，或者使用其他方法来生成唯一的节点名称。在代码中，如下进行初始化:

```
override init() {
    super.init()

    peer = MCPeerID(displayName: UIDevice.currentDevice().name)
}
```

通过UIDevice类获得设备名称。

在peer对象成功初始化后，接着进行其他对象的操作。注意peer必须最先被初始化，因为之后的对象都需要使用它。session对象:

```
override init() {
    ...

    session = MCSession(peer: peer)
    session.delegate = self
}
```

如你所见，一个session的初始化只需要一个参数，就是之前的peer。并且在初始化过程中将session对象委托给了当前类。

接着是搜索器对象:

```
override init() {
    ...

    browser = MCNearbyServiceBrowser(peer: peer, serviceType: "appcoda-mpc")
    browser.delegate = self
}
```

这个对象的初始化接受两个参数：第一个是peer. 第二个是一个在初始化后无法改变的值，指明了搜索器能搜索到的服务类型(service type)。简单地讲，它用于在大量节点中的准确识别，这样MPC就知道需要搜索什么了，而广播器也必须设置成相同的服务类型(一会儿你会见到)。在设置这个值时需要遵守两条规则：(a)不能超过15个字符(b)只能包含小写的ASCII字符、数字与连字符。如果你不符合这个规则会跳出一个runtime的异常并且app将会崩溃。

对于广播器:

```
override init() {
    ...

    advertiser = MCNearbyServiceAdvertiser(peer: peer, discoveryInfo: nil, serviceType
: "appcoda-mpc")
    advertiser.delegate = self
}
```

注意，在这里设置与之前相同的服务类型，还有一个叫discoveryInfo的参数，这个参数是一个字典，当你想给发现的其他节点发送额外信息时所需要设置的，注意，这个字典的键值都必须是字符串类型。方便起见，将这个参数设为nil。

初始化方法准备好了，这里是完整的代码:

```
override init() {
    super.init()

    peer = MCPeerID(displayName: UIDevice.currentDevice().name)

    session = MCSession(peer: peer)
    session.delegate = self

    browser = MCNearbyServiceBrowser(peer: peer, serviceType: "appcoda-mpc")
    browser.delegate = self

    advertiser = MCNearbyServiceAdvertiser(peer: peer, discoveryInfo: nil, serviceType
: "appcoda-mpc")
    advertiser.delegate = self
}
```

在结束这部分之前，来创建一个实现委托模式的协议。注意我们会声明当前在app的开发过程中所需要的所有委托方法，之后将不会再处理这个协议，直到需要它们的时候。

在自定义类的上方添加如下代码段:

```
protocol MPCManagerDelegate {
    func foundPeer()

    func lostPeer()

    func invitationWasReceived(fromPeer: String)

    func connectedWithPeer(peerID: MCPeerID)
}
```

接着会讨论上面每一个函数的作用，在下一个部分中会详细介绍。

最后，在MPCManager类中声明一个delegate对象:

```
var delegate: MPCManagerDelegate?
```

目前为止成功完成了项目的第一个重要部分，可以歇歇脚了。我想说下，现在在Xcode中提示的错误是正常情况，在实现MPC的委托方法后就会消失了。

搜索节点

MCNearbyServiceBrowserDelegate协议中有三个方法允许我们处理发现与丢失的节点，还有在搜索过程中可能发生的错误。下面要实现这些方法来继续开发这个demo app，不过这太简单了以至于不用花多少工夫。

来从第一个方法开始吧，当发现附近的节点时(换句话说，当发现其他设备时)由MPC调用。先看看实现的代码:

```
func browser(browser: MCNearbyServiceBrowser!, foundPeer peerID: MCPeerID!, withDiscov
eryInfo info: [NSObject : AnyObject]!) {
    foundPeers.append(peerID)

    delegate?.foundPeer()
}
```

首先需要着手的一个重要的行为就是在foundPeers数组中添加找到的节点(在之前已经声明了，记得吗?)，之后将这个数组作为ViewController类中列表视图的数据源，列出所有找到的节点。要这么做的话需要调用MPCManagerDelegate协议中的foundPeer委托方法，这个方法需要在ViewController类中实现(下一节中)，在那儿会重载列表数据，给用户显示最新发现的节点。

目前处理了发现一个节点时的情况，同样也需要考虑到相反的情况，需要关注若节点不再可用(可发现)时的状况。因此，来实现下一个代理方法:

```
func browser(browser: MCNearbyServiceBrowser!, lostPeer peerID: MCPeerID!) {
    for (index, aPeer) in enumerate(foundPeers){
        if aPeer == peerID {
            foundPeers.removeAtIndex(index)
            break
        }
    }

    delegate?.lostPeer()
}
```

没什么可说的，代码已经表达地很清楚了。首先找到了节点在foundPeers数组中的位置，然后移除它，这样在我们的列表里就不存在这个节点了，因此需要提醒ViewController刷新列表所显示的节点。因此需要调用lostPeer代理方法，在它的实现过程中会重载列表的数据。

最后，还有一个代理方法需要实现，这个是用来管理可能发生的错误信息，搜索不可用的情况。显然，不会在这里处理严重的错误，只是显示错误的信息，像如下代码所示来实现它:

```
func browser(browser: MCNearbyServiceBrowser!, didNotStartBrowsingForPeers error: NSEr
ror!) {
    println(error.localizedDescription)
}
```

在这一节结束前，来注意一下：使用上面的委托方法来提示ViewController类对节点变化的响应，应使用键值编码与观察者机制来追踪foundPeers数组的变化。不过由于有MPCManagerDelegate协议，就不需要去写额外的代码了。如果不需要这个协议中的其他方

法，使用KVC与KVO好一些而不是委托模式。在这里的情况，我们坚持实现有着更快速并简洁的执行过程的委托方法的做法(PS:我承认这里有问题(´･ω･｀))。

做完上面的工作后，app就能够搜索节点了。 现在，来添加一些必要的代码，显示出找到的节点吧。

显示搜索到的节点

至此这个demo应用可以发现周围的节点，并且添加(或移除)进foundPeers数组，接下来让我们在ViewController中的列表中显示它们吧。在此之前需要关注一下AppDelegate类，在那里声明MPCManager对象，这样在其他类中就可以通过app中的delegate实例来访问它了。

在工程导航栏中点击AppDelegate.swift文件打开它。在类的顶部添加如下代码:

```
var mpcManager: MPCManager!
```

进入application(application:didFinishLaunchingWithOptions:)方法，进行MPCManager的初始化:

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptio
ns: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.

    mpcManager = MPCManager()

    return true
}
```

现在，打开ViewController.swift文件，里面已经有了一个最小实现。显然还不够，需要添加更多代码，来填充与列表有关的方法，这样一切就能正常工作了。在类的顶部，从同时声明与初始化应用的delegate对象开始:

```
let appDelegate = UIApplication.sharedApplication().delegate as AppDelegate
```

在声明的同时实例化或初始化一个对象时Swift的一个特点，因为这样会使代码变得很简洁，提高开发效率。下一步就是MPCManager类的delegate设置成ViewController，在ViewDidLoad方法中添加如下行:

```
override func viewDidLoad() {
    ...

    appDelegate.mpcManager.delegate = self

}
```

在这行Xcode会显示一条错误，原因很简单:还没引入MPCManagerDelegate协议，在类的第一行如下添加一下即可:

```
class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource, MP
CManagerDelegate
```

接着来实现之前章节中使用过的两个委托方法，foundPeer和lostPeer。不管哪种情况下都需要要刷新列表即可，因此它们的实现代码很简单:

```
func foundPeer() {
    tblPeers.reloadData()
}


func lostPeer() {
    tblPeers.reloadData()
}
```

上面这部分是关键的一步，不过如果我们不“告诉”列表数据源是什么的话依旧是空白的，不做适当的修改会直接显示周围节点的名字。因此，需要从修改列表的行数与区数开始。显然，行数就是所找到的所有节点数:

```
func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return appDelegate.mpcManager.foundPeers.count
}
```

是时候显示每个节点的名字了:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -
> UITableViewCell {
    var cell = tableView.dequeueReusableCellWithIdentifier("idCellPeer") as UITableVie
wCell

    cell.textLabel?.text = appDelegate.mpcManager.foundPeers[indexPath.row].displayNam
e

    return cell
}
```

注意在上面的代码中访问了每个节点在foundPeers数组中的displayName属性。

最后来设置每个行的高度:

```
func tableView(tableView: UITableView, heightForRowAtIndexPath indexPath: NSIndexPath)
 -> CGFloat {
    return 60.0
}
```

上面的都搞定后，列表就能正常工作了。

虽然在这里已经搞定了一项重要的工作了，不过还没有考虑另一个重要的细节: 搜索功能默认是关闭的，如果不进行指示，设备是不会去搜索的。可以随时启动搜索，取决于由你的应用需求。这里，我们在app启动后立刻开启，进入viewDidLoad方法添加如下行:

```
override func viewDidLoad() {
    ...

    appDelegate.mpcManager.browser.startBrowsingForPeers()
}
```

要执行停止搜索的话，需要调用搜索器的stopBrowsingForPeers()方法。

有关搜索器的都搞定了, 接下来实现设备的广播功能。

处理广播

除了搜索，multipeer connectivity框架还允许设备对附近的节点广播自身，没有广播的话搜索就没有意义。实际上，它们是相辅相成，同等重要的。广播，意味着一个设备是否对其他设备可见。如果app中启用广播功能，则该设备就对附近的其他节点可见，不启用的话其他节点就不会发现。在这一节中来看看其中的细节，启动与禁用demo app的广播功能。

若你已经熟悉过开始工程了的话，肯定会记得在ViewController场景中有个包含一个按钮的工具栏。接着将使用这个按钮，通过实现startStopAdvertising(sender:)方法来进行开关广播功能的操作。点击按钮时会显示一个action表单，在这个action表单中有两个按钮：一个进行两种广播状态的切换，另一个控制是否打开action表单对话框。为了更有趣，使第一个按钮上的文字随着当前状态变化。

在用代码实现上述之前，先声明一个新的Bool属性，用它来判断设备是否在广播状态。确保你是在操作ViewController.swift文件，在类的顶部添加如下行:

```
var isAdvertising: Bool!
```

在viewDidLoad中会给这个变量赋值，设为true，意味着设备当前在广播自己，不过同时也需要打开这个功能，因此在viewDidLoad方法需要同时实现:

```
override func viewDidLoad() {
    ...

    appDelegate.mpcManager.advertiser.startAdvertisingPeer()

    isAdvertising = true
}
```

现在可以到action方法中实现它了。先亮出代码，之后再讲解:

```
@IBAction func startStopAdvertising(sender: AnyObject) {
        let actionSheet = UIAlertController(title: "", message: "Change Visibility", p
referredStyle: UIAlertControllerStyle.ActionSheet)

        var actionTitle: String
        if isAdvertising == true {
            actionTitle = "Make me invisible to others"
        }
        else{
            actionTitle = "Make me visible to others"
        }

        let visibilityAction: UIAlertAction = UIAlertAction(title: actionTitle, style:
UIAlertActionStyle.Default) { (alertAction) -> Void in
            if self.isAdvertising == true {
                self.appDelegate.mpcManager.advertiser.stopAdvertisingPeer()
            }
            else{
                self.appDelegate.mpcManager.advertiser.startAdvertisingPeer()
            }

            self.isAdvertising = !self.isAdvertising
        }

        let cancelAction = UIAlertAction(title: "Cancel", style: UIAlertActionStyle.Ca
ncel) { (alertAction) -> Void in

        }

        actionSheet.addAction(visibilityAction)
        actionSheet.addAction(cancelAction)

        self.presentViewController(actionSheet, animated: true, completion: nil)
    }
```

简要说下上面的实现代码都做了啥:

- 首先，使用一条信息与合适的风格来初始化了一个action表单控制器。
- 接着，根据isAdvertising的当前值，通过分配对应的值到actionTitle这个本地变量中来设置action表单中第一个按钮的标题。
- 有了正确的标题后，创建一个新的提醒行为，当用户点击第一个按钮时触发。
- 最重要的部分: 根据isAdvertising的值，停止或开始设备的广播。当然不要忘了将isAdvertising值设为相反的值。
- 为cancel按钮创建一个(空的)action。
- 两个action都添加倒了action表单控制器中。
- 最后，开启显示动画设置跳转视图。

在之后的app测试中会看到上面的action方法时如何工作的。做完上面的工作后，就可以改变设备的可被发现状态了。

邀请节点

multipeer connectivity的目的就是使两个设备间建立连接然后交换数据，我们现在只做了一半工作，仅实现了节点的发现与广播功能。下一步就是邀请节点加入到会话中，这样，两个设备间就可以进行通信了。

这里最重要的是手动来实现MPC，理解这个过程。何时来邀请找到的节点，并连接到会话中，都是由你决定的。并没有一个规定的、默认的时间，需要根据app的运行环境来选取合适的时机。比如说，当我们点击ViewController中的列表项时就会邀请对应的附近节点，也有其他情况，可能需要在搜索器发现节点后就发送邀请。何时来邀请取决于app中的实际需求。另外，还有一件同等重要的事你需要知道，发送邀请与监理连接可以在不请求用户许可的情况下进行，可以在后台执行，不过我建议你在连接其他设备或收发数据等情况时要提醒一下用户。

回到demo中，下面是这一节的目标:

1. 点击列表项时向对应名字的节点发送一个邀请。
2. 接受邀请。
3. 询问用户是否接受聊天邀请。
4. 接受或拒绝邀请

依旧在ViewController.swift文件中，来让列表来对点击事件产生响应。接着实现列表的委托方法(tableView:didSelectRowAtIndexPath:)，在这里需要执行一项重要的任务：给选择的节点发送邀请。如你所见，仅仅需要一行代码。来看看实现方法，后面会讨论它:

```
func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath)
 {
    let selectedPeer = appDelegate.mpcManager.foundPeers[indexPath.row] as MCPeerID

    appDelegate.mpcManager.browser.invitePeer(selectedPeer, toSession: appDelegate.mpc
Manager.session, withContext: nil, timeout: 20)
}
```

讲真，如果我们将选择的节点分配给一个本地变量的话就可以省去第一行代码。不管如何，第一行代码表达的更清楚，留着它吧。

使用搜索器对象的invitePeer(peerID:toSession:withContext:timeout:)方法，使用multipeer connectivity框架给选择的节点发送一个邀请。接收如下参数:

1. peerID: 想要发送邀请的节点。
2. toSession: 在MPCManager类中初始化的session对象。
3. withContext: 若要给邀请的节点发送额外数据时可以用这个参数，需要一个NSData对象。
4. timeout: 这里设置邀请者等待节点答复的最长时间。默认是30秒，这里设置为20。在一个真正的app中需要"告诉"你具体的等待时间。

现在跳转到类的顶部，导入MPC框架:

```
import MultipeerConnectivity
```

上述是第一步，接着必须实现MCNearbyServiceAdvertiserDelegate协议中的一对delegate方法，这样app就能处理一个接收到的邀请了。打开 MPCManager.swift文件。

接下来看到的方法包含一个邀请处理者(invitation handler)，用来回复发送邀请的节点。这个处理者接受两个参数，第一个是指示邀请是否被接受的bool值，第二个是session对象，对应接受邀请的情况。在demo中不会立即给邀请者一个答复，需要先询问用户是否想要接受邀请。因此必须先将邀请处理暂时储存到一个属性中，在用户响应后再回复这个邀请。之前声明过如下的方法:

```
var invitationHandler: ((Bool, MCSession!)->Void)!
```

需要储存邀请处理者，像之前说的那样，第一个delegate方法实现如下代码所示:

```
func advertiser(advertiser: MCNearbyServiceAdvertiser!, didReceiveInvitationFromPeer p
eerID: MCPeerID!, withContext context: NSData!, invitationHandler: ((Bool, MCSession!)
 -> Void)!) {
    self.invitationHandler = invitationHandler

    delegate?.invitationWasReceived(peerID.displayName)
}
```

使invitation handler与invitationHandler属性保持相同。注意这两个处理者名字相同，因此self的使用时必须的。

此外，调用了MPCManagerDelegate协议中的另一个方法。 提醒ViewController类已经接收到了一个邀请，给用户显示出一个alert控制器来询问是否聊天。之后就会看到它的实现，现在只需要关注显示节点名称就行了。

MCNearbyServiceAdvertiserDelegate协议中的第二个委托方法被用来处理无法打开广播器的情况，在控制台打印出错误信息:

```
func advertiser(advertiser: MCNearbyServiceAdvertiser!, didNotStartAdvertisingPeer err
or: NSError!) {
    println(error.localizedDescription)
}
```

现在回到ViewController.swift文件，来实现invitationWasReceived(fromPeer:)委托方法。它会显示一条信息，提醒用户聊天的对象是什么(显示节点名称)，并且提供2种选择：接受与拒绝。不论用户选择的是什么，都要使用MPCManager类的invitationHandler属性来回复邀请者。来看看如何实现的:

```
func invitationWasReceived(fromPeer: String) {
    let alert = UIAlertController(title: "", message: "\(fromPeer) wants to chat with
you.", preferredStyle: UIAlertControllerStyle.Alert)

    let acceptAction: UIAlertAction = UIAlertAction(title: "Accept", style: UIAlertAct
ionStyle.Default) { (alertAction) -> Void in
        self.appDelegate.mpcManager.invitationHandler(true, self.appDelegate.mpcManage
r.session)
    }

    let declineAction = UIAlertAction(title: "Cancel", style: UIAlertActionStyle.Cancel
) { (alertAction) -> Void in
        self.appDelegate.mpcManager.invitationHandler(false, nil)
    }

    alert.addAction(acceptAction)
    alert.addAction(declineAction)

    NSOperationQueue.mainQueue().addOperationWithBlock { () -> Void in
        self.presentViewController(alert, animated: true, completion: nil)
    }
}
```

上面是一个典型的alert控制器的实现，没什么难的地方。接受邀请时，调用mpcManager对象的invitationHandler属性，将邀请的回复设为true，并提供session对象。如果用户不想聊天，则将邀请处理者的第一个参数设为false，第二个参数为空，这种情况下不需要发送session。

又完成了一个关键步骤，接着来看看一个session的状态与当连接建立时会发生什么。

连接到会话

会话使用MPCManager类中的session对象表示，是在用multipeer connectivity时的最终目标。2个节点都连接到一个会话中时，它们就可以互相交换数据与资源、执行流传输了。一个会话有三种状态：；已连接(Connecting)、正在连接(Connecting)与断开(Not Connected)。

multipeer connectivity框架使我们对每一个状态都有控制权，通过MCSessionDelegate协议的委托方法来实现。通常这个方法的实现并不难，只需要对应每一种状态设置其相应的行为即可。在下面的代码段中，对于已连接状态，调用了一个其他的MPCManagerDelegate协议委托方法，而对于其他两种状态，仅仅在控制台显示了一条信息，这样就可以在测试过程中准确的得知当前会话的状态。粘贴下面代码的时候注意当前操作的文件是否是MPCManager.swift。

```
func session(session: MCSession!, peer peerID: MCPeerID!, didChangeState state: MCSess
ionState) {
    switch state{
    case MCSessionState.Connected:
        println("Connected to session: \(session)")
        delegate?.connectedWithPeer(peerID)

    case MCSessionState.Connecting:
        println("Connecting to session: \(session)")

    default:
        println("Did not connect to session: \(session)")
    }
}
```

使用connectedWithPeer(peerID:)委托方法通知ViewController类：设备已经与附近的一个节点(由上述的peerID参数决定)连接到了一个会话中了。

再次回到ViewController.swift文件, 来实现connectedWithPeer(peerID:)方法。在这个demo中想要在节点连接到会话时就开始聊天，因此仅需要跳转到ChatViewController场景即可。很简单:

```
func connectedWithPeer(peerID: MCPeerID) {
    NSOperationQueue.mainQueue().addOperationWithBlock { () -> Void in
        self.performSegueWithIdentifier("idSegueChat", sender: self)
    }
}
```

注意这会使两个设备(邀请者与被邀者)都执行segue，两者的会话状态都会变为已连接。

之后会去操作ChatViewController类，在操作前让我来强调一件事：我们不会去处理用户终止聊天邀请的情况，我觉得这不是很重要，交给你自己来实现吧。

一个发送数据的简便方法

在这一节我们会创建MPCManager类的一个自定义方法，用来将数据发送给其他节点。实际上是调用MCSession类中一个负责发送数据的特殊方法，不过首先需要准备并配置好这个方法所需要的所有参数。创建这个自定义方法的原因是因为避免在多个地方进行配置，一次搞定。

先给你实现的方法 (确保打开了MPCManager.swift文件):

```
func sendData(dictionaryWithData dictionary: Dictionary<String, String>, toPeer target
Peer: MCPeerID) -> Bool {
    let dataToSend = NSKeyedArchiver.archivedDataWithRootObject(dictionary)
    let peersArray = NSArray(object: targetPeer)
    var error: NSError?

    if !session.sendData(dataToSend, toPeers: peersArray, withMode: MCSessionSendDataM
ode.Reliable, error: &error) {
        println(error?.localizedDescription)
        return false
    }

    return true
}
```

上面的sendData(data:toPeers:withMode:error:)就是所调用的MPCSession方法，它接受如下参数:

- data: 一个NSData对象，实际发送的数据。
- toPeers: 一个接收数据的节点数组 (NSArray)。
- withMode: 数据发送模式，有两种模式: 可靠与不可靠。如果没收到的数据不会引起任何问题，这种数据不重要的情况下可以使用第二种模式。
- error: 包含可能发生error的一个NSError对象。

现在来看看上面的实现代码发生了什么，如你所见，这个方法有两个参数：(a)一个字典对象(b)目标节点。首先，使用NSKeyedArchiver类对字典进行归档，将其转化成一个NSData对象，然后定义一个仅包含目标节点的单元素peersArray数组。声明error变量后，将上面提供的变量作为输入参数，调用session对象的数据发送方法。

若在发送数据时有错误，会将其显示出来然后返回false，否则返回true，意味着一切正常。

上面这个方法会在下一节用到。

在节点间发送数据

在聊天会话的过程中节点发送与接收的所有消息，都会显示在ChatViewController的列表中。最靠后显示的消息总是时刻最近的那条，每当一条消息被发送与接收时都会使列表重载。

你可能猜到了，会使用一个数组来储存所有消息，显然这个数组会被作为列表的数据源。重要的是，这个数组中的每一个对象都是一个包含字符串类型键值的字典。为何是字典?因为需要为每个发送或接收的消息分配一对数据：消息的发送方与消息本身。当我们的设备是消息发送方时，设备中的消息发送方就会被设为"self"，我们的节点名就会发送给其他设备。

回到编码的工作中，现在有一些事情要做了。第一步就是要声明并初始化消息数组(列表的数据源), 之后会声明并初始化一个应用的delegate对象，因此会访问APPDelegate类的mpcManager属性。打开ChatViewController.swift文件，在类顶部添加如下两行:

```
var messagesArray: [Dictionary<String, String>] = []

let appDelegate = UIApplication.sharedApplication().delegate as AppDelegate
```

messagesArray初始化为一个空数组，我们不打算将ChatViewController设为mpcManager的委托对象，将使用另一种途径来从MPCManager类中获取消息。

在聊天时，每当一个新消息完成编辑并且发送按钮被按下时会触发一些行为，下面这些是我们需要做的:

1. 隐藏键盘。
2. 根据消息创建一个字典，调用上一节的自定义方法将消息发送给其他节点。
3. 创建另一个字典，将发送方与消息作为其内容，然后储存进messagesArray数组中。
4. 刷新列表。
5. 消息发送后清空输入栏。

上述这些发生在UITextFieldDelegate协议中的委托方法textFieldShouldReturn(textField:)中。如果看看viewDidLoad方法，会发现ChatViewController类已经被设为了textfield的委托对象。

代码实现如下:

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    textField.resignFirstResponder()

    let messageDictionary: [String: String] = ["message": textField.text]

    if appDelegate.mpcManager.sendData(dictionaryWithData: messageDictionary, toPeer:
appDelegate.mpcManager.session.connectedPeers[0] as MCPeerID){

        var dictionary: [String: String] = ["sender": "self", "message": textField.tex
t]
        messagesArray.append(dictionary)

        self.updateTableview()
    }
    else{
        println("Could not send data")
    }

    textField.text = ""

    return true
}
```

需要注意的点：调用了自定义方法sendData(dictionaryWithData:toPeer:)，输入之前创建的messageDictionary。有趣的是我们使用appDelegate.mpcManager.session.connectedPeers[0]对象来指明目标节点，需要说明的是MCSession类包含一个叫connectedPeers的数组属性，每个连接到我们设备的节点都会被记录下来。在我们的代码中，由于已经知道只有1个节点会加入到会话中， 因此直接访问数组的第一个元素即可。

如果数据被成功发送了，则为发送方与消息准备一个新的字典。如果是我们的消息，则sender被设为"self"，接着使用messagesArray的append方法将字典添加到数组中。最后调用updateTableview方法刷新列表，这是个自定义方法，一会儿会实现它。

如果有错误发生，会在控制台显示一条信息。不管发生什么，在上述方法的最后都会清空输入栏。

要编写的updateTableview方法有两个目的:首先是刷新列表，显示新的消息。其次，自动滚动到列表底部，这样最新的消息就能时刻显示了 。代码如下:

```
func updateTableview(){
    self.tblChat.reloadData()

    if self.tblChat.contentSize.height > self.tblChat.frame.size.height {
        tblChat.scrollToRowAtIndexPath(NSIndexPath(forRow: messagesArray.count - 1, in
Section: 0), atScrollPosition: UITableViewScrollPosition.Bottom, animated: true)
    }
}
```

当列表的content size大于列表的框体高度时，则必须要滚动，使用上面的方法实现。

现在，在类的顶部导入multipeer connectivity框架， 修复在textFieldShouldReturn(textField:)中的错误:

```
import MultipeerConnectivity
```

在这一节结束前还需要做一件事，需要完善列表相关的方法。首先需要将messagesArray数组中已存在的元素个数分配给行数:

```
func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return messagesArray.count
}
```

在tableView(tableView:cellForRowAtIndexPath:)方法中需要检查发送方的消息。如果发送方的值是"self"，则将子标签设为紫色，会显示消息的前缀"I said:"。否则设为橘黄色，显示"X said:"，X是其他节点的名称。代码如下:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -
> UITableViewCell {
    var cell = tableView.dequeueReusableCellWithIdentifier("idCell") as UITableViewCell


    let currentMessage = messagesArray[indexPath.row] as Dictionary<String, String>

    if let sender = currentMessage["sender"] {
        var senderLabelText: String
        var senderColor: UIColor

        if sender == "self"{
            senderLabelText = "I said:"
            senderColor = UIColor.purpleColor()
        }
        else{
            senderLabelText = sender + " said:"
            senderColor = UIColor.orangeColor()
        }

        cell.detailTextLabel?.text = senderLabelText
        cell.detailTextLabel?.textColor = senderColor
    }

    if let message = currentMessage["message"] {
        cell.textLabel?.text = message
    }

    return cellc
}
```

上面的方法中没什么难的地方，就不讨论它了。

需要关注一个列表行的高度，很明显我们不能确定高度是多少，因为消息的长度是变化的，每行的高度都需要动态变化。因此使用一个iOS 8的特性，叫做self sizing cells(ps:早知道这个就好了...)，可以看看Simon的这个教程。将列表栏的文本标签的行数设为0，然后在viewDidLoad方法中设置以下两个属性:

```
tblChat.estimatedRowHeight = 60.0
tblChat.rowHeight = UITableViewAutomaticDimension
```

上述代码在viewDidLoad中已经有了，交给iOS处理吧。

把上面的搞定后就可以进行接收数据的工作了。

接收数据

现在app可以发送消息了，现在需要处理接收数据的环节了。接着要操作MPCManager与ChatViewController类。首先，来实现一个MPCSessionDelegate协议中的一个新方法。

打开MPCManager.swift文件，添加如下方法:

```
func session(session: MCSession!, didReceiveData data: NSData!, fromPeer peerID: MCPeerID!) {
    let dictionary: [String: AnyObject] = ["data": data, "fromPeer": peerID]
    NSNotificationCenter.defaultCenter().postNotificationName("receivedMPCDataNotification", object: dictionary)
}
```

只包括仅仅两行却至关重要的代码。首先将接收到的数据与发送方节点添加到一个字典中，接着提交一个名为receivedMPCDataNotificaton的通知(NSNotification)，在ChatViewController中监听这个通知，作出适当的处理，在列表中显示发送方的名称与消息。在上一节中我说过不要将ChatViewController设为MPCManager的委托对象，会通过另一种途径从这个类中获取消息，这里的不同的途径就是上面我们提交的通知。

现在让我们回到ChatViewController.swift文件中，在viewDidLoad方法中来监听上面的通知。这很简单，只需要下面几行代码即可:

```
override func viewDidLoad() {
    ...

    NSNotificationCenter.defaultCenter().addObserver(self, selector: "handleMPCReceivedDataWithNotification:", name: "receivedMPCDataNotification", object: nil)
}
```

这么做了后，每当收到新数据时都会提交一个通知，ChatViewController会做出响应。

还剩最后一步：实现handleMPCReceivedDataWithNotification(notification:)，当检测到通知时会调用它。

在实现它之前，先来大概说说在其中发生了什么:

- 第一步，需要得到通知中带有的字典，“提取”其中的数据与节点。
- 将数据对象转换为字典，即可访问其中的数据。
- 设置一个规则，指明聊天终止的一个特殊短语，这个短语即为“end_chat”消息。
- 若消息不是上述的特殊值，则创建一个包含发送方名称与消息的字典，将其添加进messagesArray数组中，并刷新列表。
- 若消息表示聊天的终点，即提醒用户另一个节点终止了聊天，则会返回当前视图控制器。在下一节中会编写这一步的代码。

来看看上述代码的实现，代码中的额外注释会帮助你理解:

```
func handleMPCReceivedDataWithNotification(notification: NSNotification) {
    // Get the dictionary containing the data and the source peer from the notificatio
n.
    let receivedDataDictionary = notification.object as Dictionary<String, AnyObject>

    // "Extract" the data and the source peer from the received dictionary.
    let data = receivedDataDictionary["data"] as? NSData
    let fromPeer = receivedDataDictionary["fromPeer"] as MCPeerID

    // Convert the data (NSData) into a Dictionary object.
    let dataDictionary = NSKeyedUnarchiver.unarchiveObjectWithData(data!) as Dictionary
<String, String>

    // Check if there's an entry with the "message" key.
    if let message = dataDictionary["message"] {
        // Make sure that the message is other than "_end_chat_".
        if message != "_end_chat_"{
            // Create a new dictionary and set the sender and the received message to
it.
            var messageDictionary: [String: String] = ["sender": fromPeer.displayName,
"message": message]

            // Add this dictionary to the messagesArray array.
            messagesArray.append(messageDictionary)

            // Reload the tableview data and scroll to the bottom using the main threa
d.
            NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
                self.updateTableview()
            })
        }
        else{

        }
    }
}
```

准备好了。现在若有消息来到时，app就会在列表中显示它了。

终止聊天

离应用中功能全部实现的工作所剩无几了，其中一个就是终止聊天，发生在其中一个节点想要终止或会话不是已连接状态。

在ChatView Controller场景中最顶部的工具栏上有一个按钮，点击后会调用endChat(sender:)方法。使用这个方法给其他节点发送终止消息告诉它们聊天结束，然后返回到上一个视图控制器。当然，这个消息即为上一节所说的“_end_chat”短语。

来看看具体实现:

```
@IBAction func endChat(sender: AnyObject) {
    let messageDictionary: [String: String] = ["message": "_end_chat_"]
    if appDelegate.mpcManager.sendData(dictionaryWithData: messageDictionary, toPeer:
appDelegate.mpcManager.session.connectedPeers[0] as MCPeerID){
        self.dismissViewControllerAnimated(true, completion: { () -> Void in
            self.appDelegate.mpcManager.session.disconnect()
        })
    }
}
```

如你所见，在这里dismiss了视图控制器然后使用MCSession类的disconnect()方法断开了与节点间的会话。在跳转动画完成后才断开会话连接，给了会话足够的存活时间。

在最后一节中实现了自定义方法handleMPCReceivedDataWithNotification(notification:)中的部分代码，为何我会说“部分”，因为还没有添加任何处理当发送终止聊天消息时的代码。是时候添加了，在做之前需要注意，要给用户显示一个alert控制器提醒他其它节点结束了聊天，在那之后，我们会从会话中断开并dismiss视图控制器。这里是代码中缺失的部分:

```
func handleMPCReceivedDataWithNotification(notification: NSNotification) {
    ...

    // Check if there's an entry with the "message" key.
    if let message = dataDictionary["message"] {
        ...
        else{
            // In this case an "_end_chat_" message was received.
            // Show an alert view to the user.
            let alert = UIAlertController(title: "", message: "\(fromPeer.displayName)
 ended this chat.", preferredStyle: UIAlertControllerStyle.Alert)

            let doneAction: UIAlertAction = UIAlertAction(title: "Okay", style: UIAler
tActionStyle.Default) { (alertAction) -> Void in
                self.appDelegate.mpcManager.session.disconnect()
                self.dismissViewControllerAnimated(true, completion: nil)
            }

            alert.addAction(doneAction)

            NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
                self.presentViewController(alert, animated: true, completion: nil)
            })
        }
    }
}
```

当其它节点关闭app时也会导致聊天被终止，或者由于某些原因两设备间的连接丢失，这时需要考虑的情况。要做的就是在MPCManager.swift类的browser(browser:lostPeer:)代理方法中添加一个新的通知，然后还需要监听并处理它，就像之前的那个通知一样。不过这次我就不给你看具体的实现了，作为练习麻烦你自己动手完成吧。

快成功了，再进行一些小小的增补就可以测试了。

最后修整

离测试app只剩一步了，还有一些事没有做。需要在MPCManager类中定义一些MCSessionDelegate协议中的委托方法，虽然用不上。这样做了的话，一些Xcode中的错误就会消失了。

不要犹豫，打开MCPManager.swift文件copy下面的方法即可:

```
func session(session: MCSession!, didStartReceivingResourceWithName resourceName: String!, fromPeer peerID: MCPeerID!, withProgress progress: NSProgress!) { }

func session(session: MCSession!, didFinishReceivingResourceWithName resourceName: String!, fromPeer peerID: MCPeerID!, atURL localURL: NSURL!, withError error: NSError!) {
 }

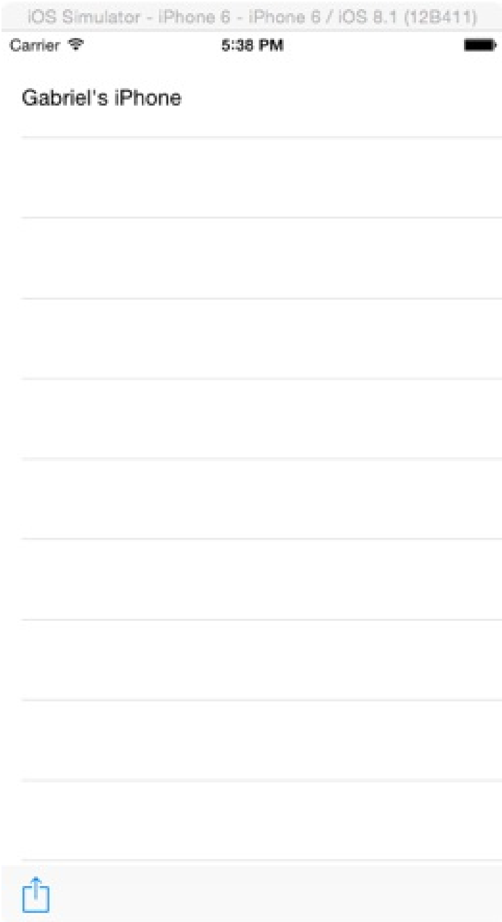
func session(session: MCSession!, didReceiveStream stream: NSInputStream!, withName streamName: String!, fromPeer peerID: MCPeerID!) { }
```

测试app

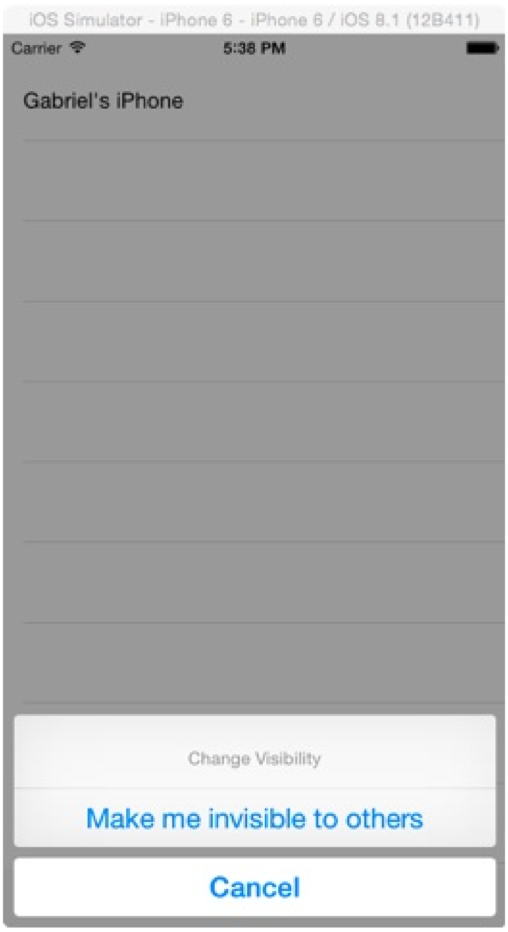
可以在两台设备(至少)上运行demo，也可以在一台设备与一个虚拟机上运行。首先通过开关app中的广播功能来操作搜索器与广播器，然后选择一个节点来开始聊天，发送、接收消息，最后终止对话。如果你完成了测试，可以改改代码，添加一些你想要的功能。

我要展示一些app的截图，注意我是在iPhone模拟器上运行的。

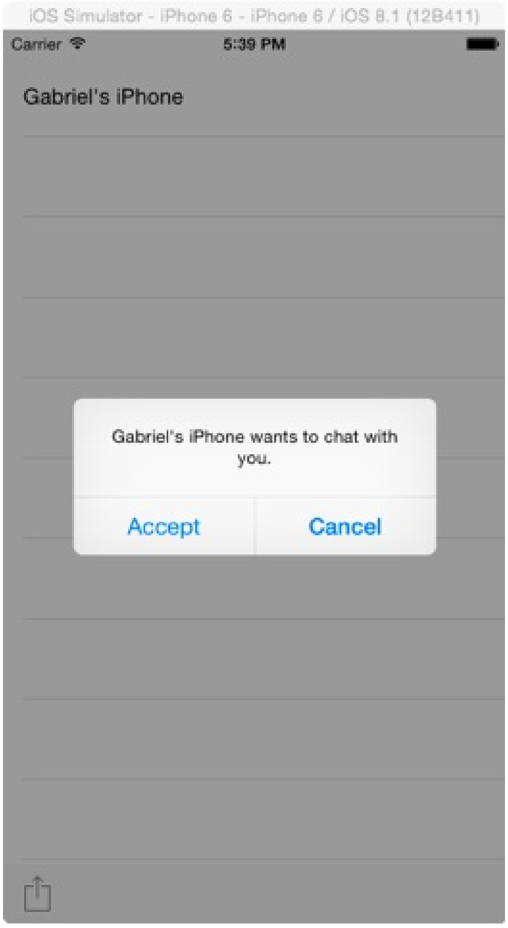
发现一个附近的节点:


iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.1 (12B411)
Carrier
5:38 PM
Gabriel's iPhone

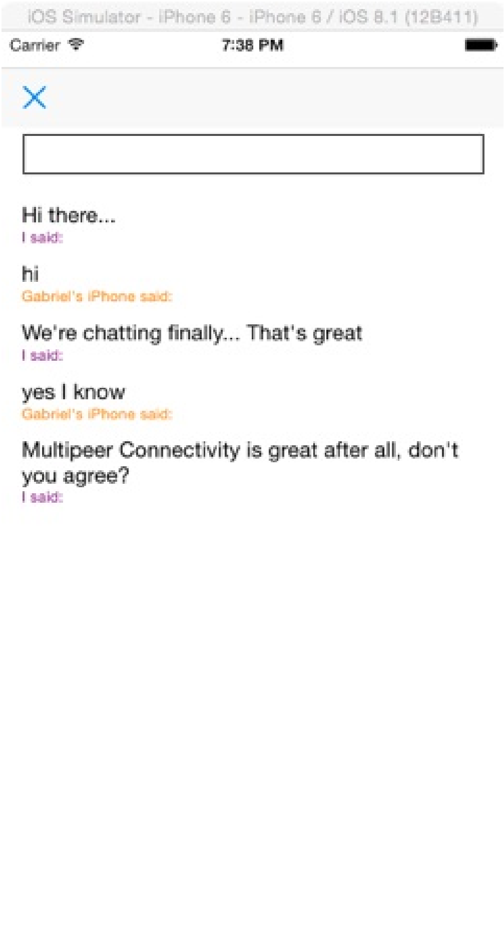
关闭广播器:


iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.1 (12B411)
Carrier
5:38 PM
Gabriel's iPhone
Change Visibility
Make me invisible to others
Cancel

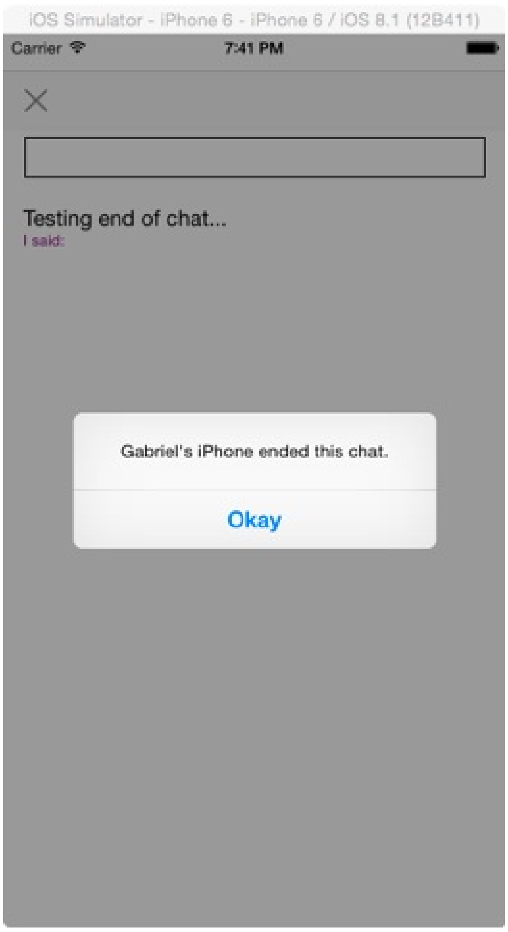
询问用户是否聊天:



进行聊天:


iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.1 (12B411)
Carrier
7:38 PM
Hi there...
I said:
hi
Gabriel's iPhone said:
We're chatting finally... That's great
I said:
yes I know
Gabriel's iPhone said:
Multipeer Connectivity is great after all, don't you agree?
I said:

结束聊天:


iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.1 (12B411)
Carrier
7:41 PM
Testing end of chat...
I said:
Gabriel's iPhone ended this chat.
Okay

可以在这里下载最后的工程。

Using Sleep Analysis in HealthKit with Swift

使用Swift与HealthKit的睡眠分析

> - 原文链接：Using Sleep Analysis in HealthKit with Swift
> - 原文作者：ANUSHK MITTAL
> - 译者：yrq110

睡眠革命是一个当今很流行的话题，用户对这个有着很大的兴趣，不论是对他们自己的睡眠时间，还是对通过分析采集的数据所揭示的周期性睡眠趋势。伴随着硬件中技术的提升，尤其是手机，给予了这个关注度日益增长的问题一个全新的视角。

苹果提供了个很cool的手段，以一种安全的方式去处理用户的个人健康数据，将数据安全的存放在了内置的Health app中。你不仅可以使用HealthKit来做健身app，也可以获取睡眠分析的数据。

在这篇教程中会简要介绍HealthKit框架，并告诉你如何构建一个简单的睡眠分析app。

目录

简介

HealthKit提供了一种在加密数据库中存储数据的结构，叫做HealthKit store，可以使用HKHealthStore类来访问对应的数据库。iPhone和Apple Watch有各自的HealthKit store。Apple Watch与iPhone的健康数据是同步的，不过Apple Watch会定期清理旧数据以节省空间。HealthKit与Health app在iPad上不可用。

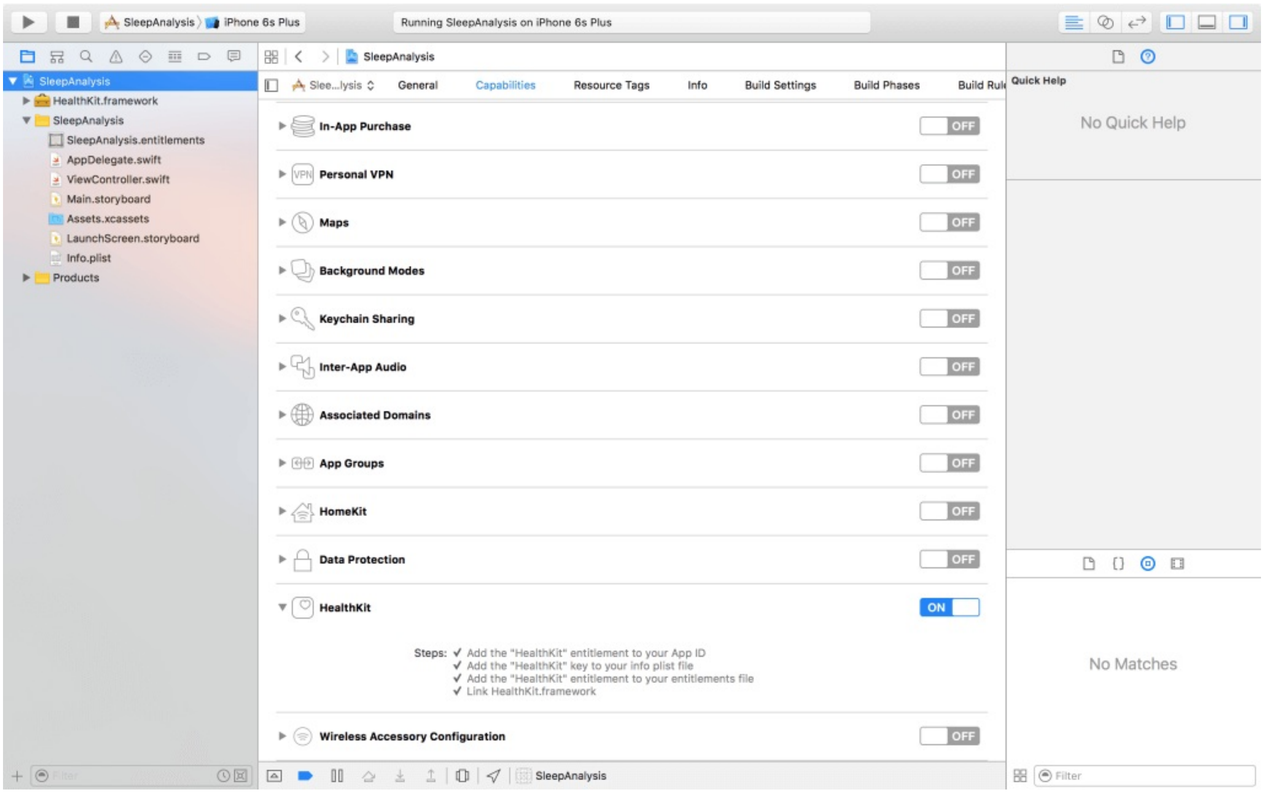
如果你想做一个基于健康数据的iOS或watchOS的app，HealthKit将会是一个有力的工具。HealthKit用于管理多个来源的数据，根据用户的喜好来融合整理这些数据，App也可以访问每一个来源的原始数据，并且自己来处理数据。这些数据不仅可以进行人体测量，用来健身、控制营养，也可以用来睡眠分析。

在文章中剩下的部分，将会展示如何利用HealthKit框架来存储与访问iOS上的睡眠分析数据，对于watchOS应用来说也是同样的方法。请注意这篇教程是基于Swift 2.0与Xcode 7所写，请确保你使用的是Xcode 7(或更高)。

在进行下一步之前，先下载开始工程然后解压它。我已经创建好了UI，当你运行工程时，会出现一个计时器的UI，点击start按钮时计时器会开始计时。

使用HealthKit框架

应用的目标就是储存睡眠分析信息，使用Start与Stop按钮来检索。为了使用HealthKit，你必须先在app bundle中开启HealthKit功能。在工程中，点击当前工程target -> capabilities 打开HealthKit。



接下来，先在ViewController类中创建一个HKHealthStore的实例，代码如下:

```
let healthStore = HKHealthStore()
```

后面会使用这个HKHealthStore实例来访问HealthKit store。

如前所述，HealthKit允许用户控制他们的健康数据，因此首先你需要得到用户许可才可以对用户的睡眠分析数据进行访问的权限(读/写)。首先import内置的HealthKit框架，然后如下修改ViewDidLoad方法:

```
override func viewDidLoad() {
    super.viewDidLoad()

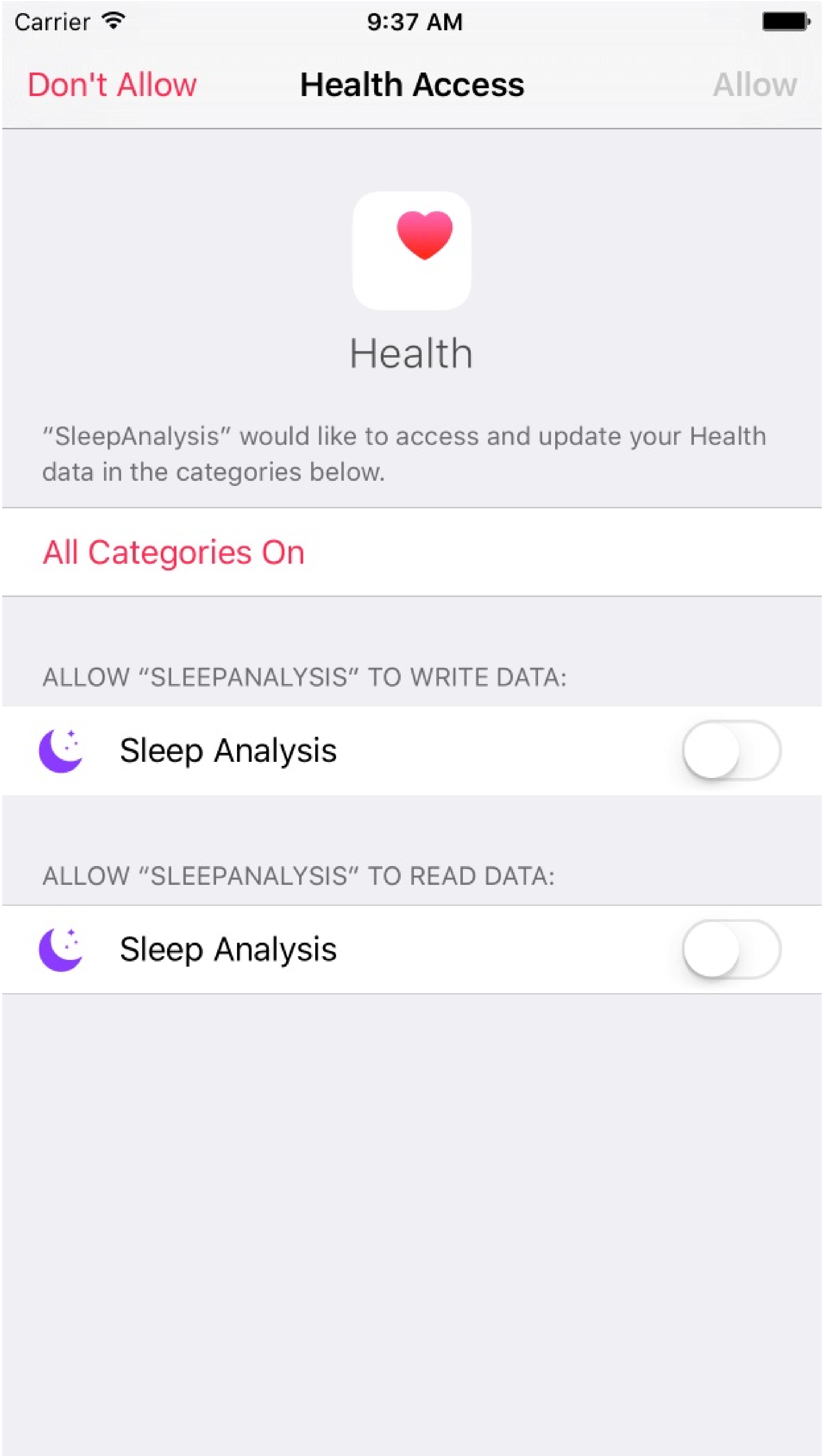
    let typestoRead = Set([
        HKObjectType.categoryTypeForIdentifier(HKCategoryTypeIdentifierSleepAnalysis)!
        ])

    let typestoShare = Set([
        HKObjectType.categoryTypeForIdentifier(HKCategoryTypeIdentifierSleepAnalysis)!
        ])

    self.healthStore.requestAuthorizationToShareTypes(typestoShare, readTypes: typesto
Read) { (success, error) -> Void in
        if success == false {
            NSLog(" Display not allowed")
        }
    }
}
```

这段代码会提示用户允许还是拒绝请求的权限。在闭包中，可以处理成功或失败情况下的行为，得到最终的结果。没必要使用户允许app所有的请求权限，不过在app中必须优雅地处理失败的情况。

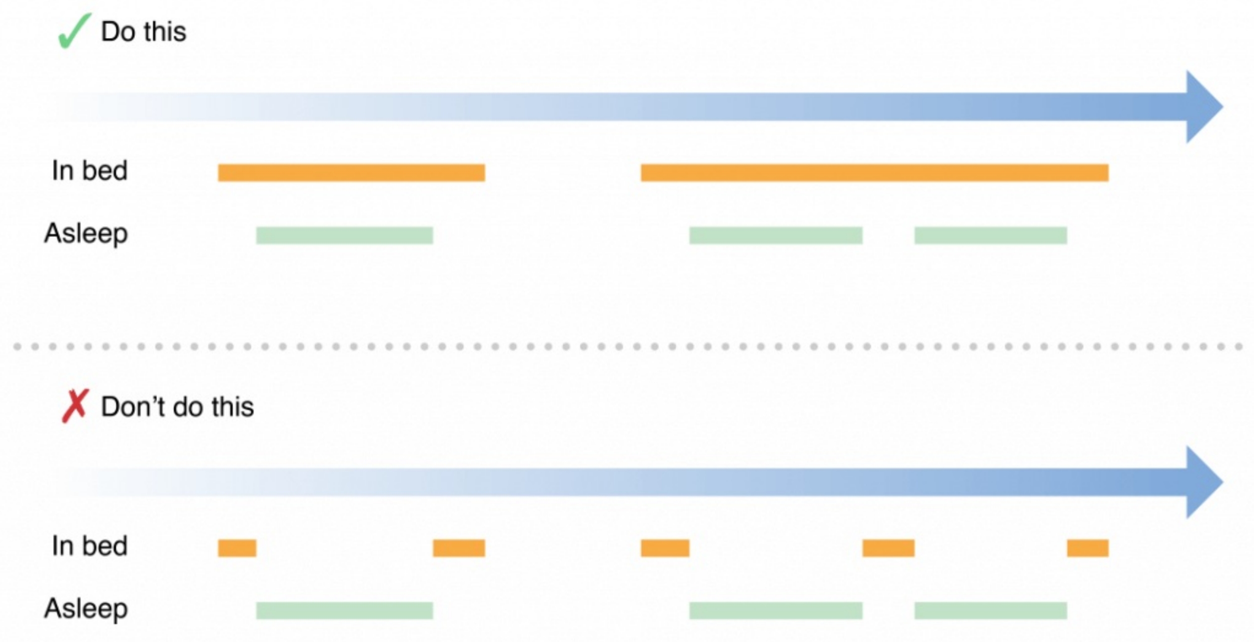
不过为了测试，你必须选择“Allow”去允许app获得访问设备中健康数据的权限。


Carrier
9:37 AM
Don't Allow
Health Access
Allow
Health
"SleepAnalysis" would like to access and update your Health data in the categories below.
All Categories On
ALLOW "SLEEPANALYSIS" TO WRITE DATA:
Sleep Analysis
ALLOW "SLEEPANALYSIS" TO READ DATA:
Sleep Analysis

写入睡眠分析数据

首先我们如何来检索睡眠分析数据呢？根据苹果的文档，每个睡眠分析的样本只能对应一种状态，为了同时表现用户在上床与熟睡两种状态，HealthKit使用多于2个的样本。比较开始与结束时的这些样本，app可以计算出多种统计数据

- 进入睡眠所花费的时间
- 用户实际睡眠的时间百分比
- 用户在床上醒来的次数
- 上床与睡眠所花费的总时间



简要的来说，需要执行以下步骤来将睡眠分析数据储存进HealthKit store中:

1. 定义两个NSDate对象，对应开始时间与结束时间
2. 使用HKCategoryTypeIdentifierSleepAnalysis创建一个HKObjectType的实例
3. 创建一个HKCategorySample类型的新对象，通常会使用定义好的类别样本去记录睡眠数据，每个样本都可以表现出用户在单位时间内床上或睡眠时的数据，需要创建在时间上重叠的床上样本与睡眠样本。
4. 最后，使用HKHealthStore的saveObject保存对象

> 关于样本类型，可以看下HealthKit Constants Reference。

下面这段代码用Swift实现了上面的步骤，同时保存了在床上和睡眠两种状态的睡眠分析数据。请在 ViewController类中插入这个方法:

```
func saveSleepAnalysis() {

    // alarmTime and endTime are NSDate objects
    if let sleepType = HKObjectType.categoryTypeForIdentifier(HKCategoryTypeIdentifier
SleepAnalysis) {

        // we create our new object we want to push in Health app
        let object = HKCategorySample(type:sleepType, value: HKCategoryValueSleepAnaly
sis.InBed.rawValue, startDate: self.alarmTime, endDate: self.endTime)

        // at the end, we save it
        healthStore.saveObject(object, withCompletion: { (success, error) -> Void in

            if error != nil {
                // something happened
                return
            }

            if success {
                print("My new data was saved in HealthKit")

            } else {
                // something happened again
            }

        })


        let object2 = HKCategorySample(type:sleepType, value: HKCategoryValueSleepAnal
ysis.Asleep.rawValue, startDate: self.alarmTime, endDate: self.endTime)

        healthStore.saveObject(object2, withCompletion: { (success, error) -> Void in
            if error != nil {
                // something happened
                return
            }

            if success {
                print("My new data (2) was saved in HealthKit")
            } else {
                // something happened again
            }

        })

    }

}
```

当我们需要保存睡眠分析数据到HealthKit中时就可以调用它。

读取睡眠分析数据

为了读取睡眠分析数据，需要创建一个查询对象。首先应定义HKCategoryTypeIdentifierSleepAnalysis中需要的HKObjectType类别。也许你想使用谓词过滤器去过滤检索得到的数据，检索的数据是startDate与endDate这两个NSDate对象对应的时间范围内的数据，也可能想创建一个用于整理检索查询结果并选择期望结果的sortDescripter。

检索睡眠分析数据的代码应如下所示:

```
func retrieveSleepAnalysis() {

    // first, we define the object type we want
    if let sleepType = HKObjectType.categoryTypeForIdentifier(HKCategoryTypeIdentifier
SleepAnalysis) {

        // Use a sortDescriptor to get the recent data first
        let sortDescriptor = NSSortDescriptor(key: HKSampleSortIdentifierEndDate, asce
nding: false)

        // we create our query with a block completion to execute
        let query = HKSampleQuery(sampleType: sleepType, predicate: nil, limit: 30, so
rtDescriptors: [sortDescriptor]) { (query, tmpResult, error) -> Void in

            if error != nil {

                // something happened
                return

            }

            if let result = tmpResult {

                // do something with my data
                for item in result {
                    if let sample = item as? HKCategorySample {
                        let value = (sample.value == HKCategoryValueSleepAnalysis.InBed
.rawValue) ? "InBed" : "Asleep"
                        print("Healthkit sleep: \(sample.startDate) \(sample.endDate)
- value: \(value)")
                    }
                }
            }
        }

        // finally, we execute our query
        healthStore.executeQuery(query)
    }
}
```

这段代码查询HealthKit中的数据得到所有睡眠分析数据，并使用降序排列。每一次查询都会打印出开始时间、结束时间与对应的状态值。这里将限制参数设成了30，因此会检索最后30个样本，也可以使用谓词过滤的方法去选择你想要的开始与结束时间。

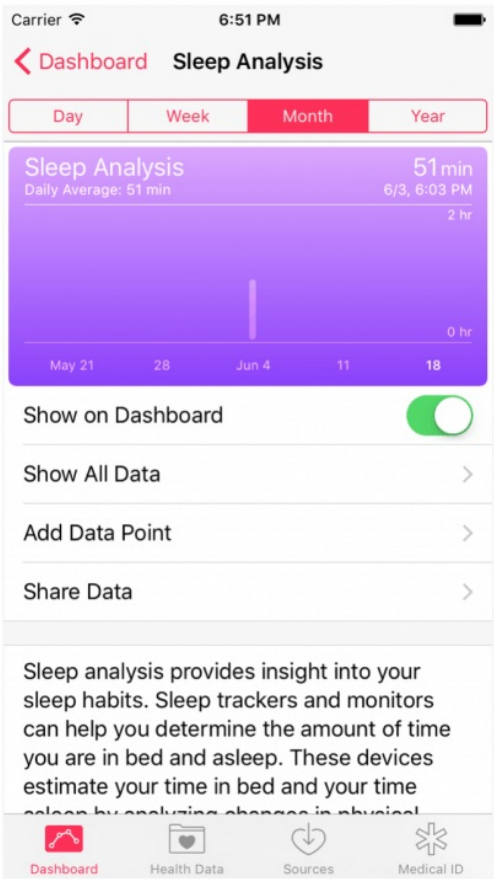
App测试

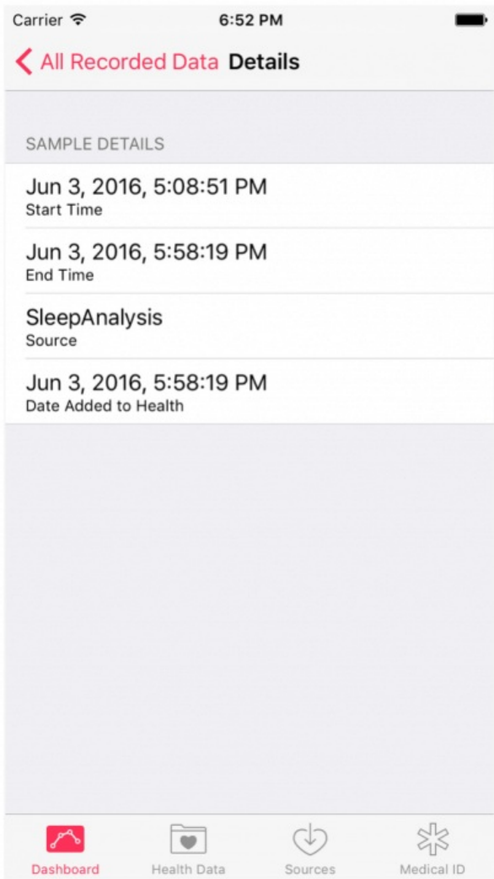
在demo中我使用了一个计时器来表现时间的流逝，点击start按钮启动计时器。点击开始与结束按钮就会创建对应的NSDate对象，保存这个时间段内的睡眠分析数据。在停止的方法内可以调用saveSleepAnalysis()与retrieveSleepAnalysis()方法来保存与取得睡眠数据。

```
@IBAction func stop(sender: AnyObject) {
    endTime = NSDate()
    saveSleepAnalysis()
    retrieveSleepAnalysis()
    timer.invalidate()
}
```

在app中你也许会修改保存不同状态数据的开始与结束时间。

做出修改后，启动demo app与计时器，让它运行几分钟然后点击Stop按钮。打开Health app，就会看到睡眠数据了。





一些HealthKit App的建议

HealthKit为app开发者提供了一个可以分享并能轻松访问到用户数据的公共平台，避免了数据的复制与不一致的问题。在苹果的审核指南中，对于使用HealthKit时对读取/写入权限的请求没有明确说明的app可能会被拒绝。

存储虚假或不正确数据的Health App也会被拒绝，意味着你不能使用自己的算法去计算不同的健康数据，像这篇教程中的睡眠分析。应读取并使用内置传感器数据，操作各种参数避免计算虚假的数据。

可以从这里下载完整的Xcode工程。

Documenting Your Swift Code in Xcode Using Markdown

使用Markdown文档化Xcode中的Swift代码

> - 原文链接 : Documenting Your Swift Code in Xcode Using Markdown
> - 原文作者 : GABRIEL THEODOROPOULOS
> - 译者 : yrq110

在Xcode 7所包含的所有特性中，有一个新的革命毋庸置疑：以一个更好的方式去书写代码文档。开发者可以使用强大的Markdown语句写出含有多种文本样式的文档，结合特定的关键字标识出特殊部分(比如参数、函数结果等等)，会有意想不到的结果。新的启用Markdown的文档风格有如下优点：提供了更高等级的自定义文本能力，更加灵活，当然也更加有趣。不过如果你对老式的文档风格感兴趣，可以看看我们之前的教程。

撰写代码文档对每个开发者来说都是一项重要的任务，虽然看上去这会拖慢开发的进度，不过的确是开发中的一部分。我不觉得为每一个属性、函数、类、结构体与其他项目中存在的项撰写合适的、便于理解的文档是件容易的事情。可以通过以下要点来书写文档：

- 描述属性、函数与类的作用、目的、细节等，高亮函数特定的条件、用例或者规则。
- 高亮方法的输入与输出(参数与返回值)。
- 在初始实现后，当你重新审视代码时，函数与属性的作用一目了然。
- 使其他开发者理解如何使用你的代码或库。
- 使用工具来产生专业的手册(比如说Jazzy).

可以通过三种方式来预览与访问在Xcode中书写的代码文档:

- 按住Option/Alt点击属性名、方法等、类名等，快速预览窗会在名称附近弹出。
- 将光标放置在属性名、方法名或类名上，打开快速帮助。
- 使用可以生成手册的第三方工具。比如说Jazzy是个不错的工具，之后会说它。使用它的话，你的文档就可以编译成网页，整个文档可以组成一个脱机的网站，保存在工程文件夹中的一个子文件夹。

代码文档不是一成不变，它需要是灵活的，根据实体中(属性、方法、类、结构体、枚举)产生的修改来改变。若当你实现了一个新实体时没有添加文档则之后你也不会添加的。因此，养成一个在适当的时机添加代码文档的习惯吧，分配一些时间去做，这是值得的。

目录

Markdown语句基础

为了充分利用新的文档风格，了解一下Markdown语句的相关知识是很重要的，如果你已经知道了就跳过这一节吧。可以在网上找到很多关于Markdown的资料，比如说这儿和这儿有一些不错的阅读资料。

你可以找到其他有关Markdown语句的资源，我这里只要求最低要求的基础语句，提供完整全面的引导教程不是我所关注的，将注意力放在最常用标注语句的表现上。

你可能知道(或者你刚学了) Markdown语句是通过使用特殊的字符来格式化文本、添加资源(例如链接或图像)与标注特殊区域或文本(例如有序或无序列表，代码块)。这些字符很好记，可以通过反复使用或查询下面的列表来加深记忆。当你习惯了Markdown语句后会发现很有趣，通过使用合适的表及其可以产生不同格式的文档：比如HTML网页、PDF文档等等。说到HTML，Markdown支持内联的HTML，意味着你可以直接在文本中加入HTML标签，然后会进行适当的渲染。不过HTML并不是Markdown的本质，因此只关注Markdown本身的语句。

下面这个列表包含了常用的Markdown语句:

- #text#: 将文本设置为header，与HTML中的 `<H1>` 等价。两个#字符相等于 `<H2>` header，直到 `<H6>` 同理，后面的#字符是可选的。
- **text**: 返回星号中间加粗的文本。
- *text*: 返回星号中间斜体的文本。
- * text: 创建一个无序列表项(注册在星号与文本间有一个空格)，也可以使用“+”或“-”符号。
- 1. text: 创建一个有序列表项 (数字列表)。
- [linked text](http://some-url.com\): 当文本转换为一个链接。
- > text: 创建一个块引用
- 使用4个空格或tab标记书写的代码块，与HTML中的 `<pre>``</pre>` 标签，如果想添加缩写需要添加额外的4个空格或1个tab字符。
- 如果你不想被空格和tab搞混，可以将代码写在“`” 字符中。比如说`var myProperty`会显示成: `var myProperty` 。
- 另一种创建代码块的方式是添加4个(“`”)符号，然后开始代码的书写，在代码的最后一行添加另外4个(“`”)。
- 使用反斜杠禁用Markdown语句中特殊字符的渲染。例如：**/this/**就不会显示成加粗的字体。

上述是需要了解的Markdown语句中最重要的元素，其实还有更多，给你介绍的这些足够入门了，想了解更多细节的话请自己去查阅更多的资料。

对Markdown产生兴趣后，寻找一个免费的编辑器(在线的或者是Mac软件)，来上手尝试一下。如果你找的编辑器支持实时预览的话，就可以观察到你的输入会直接转换成HTML，立即看到输出。

> 编者的话: 可以尝试如下编辑器StackEdit (在线), Typora, Macdown, Focused 与 Ulysses。

使用**Markdown**

在Swift中对各类实体编写文档需要遵守一些特性的规则，可以对属性(变量与常量)、方法、函数、类、结构体、结构体、枚举、协议、扩展与其他代码结构或实体编写文档。文档块需要在实体的声明或头文件后来编写，使用3个斜杠或将代码块包含在如下的结构中:

```
/**

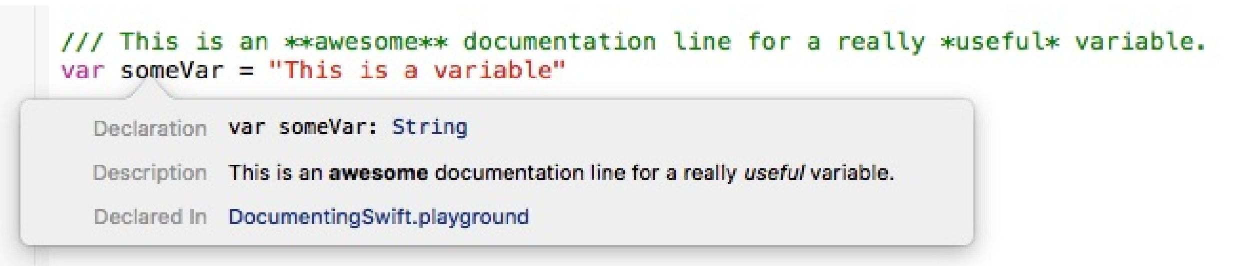
*/
```

双斜杠(//)会被Xcode无视，不会渲染任何文档。在你的代码块(一个方法的body)中使用双斜杠，而不是在编写整个实体的文档时。

来看一个使用Markdown语句的简单例子，在下面的代码中文档化一个简单的属性。建议你在Xcode中新建一个Playground，在里面把玩示例。

```
/// This is an **awesome** documentation line for a really *useful* variable.
var someVar = "This is a variable"
```

效果如下:



注意 “awesome” 被显示成了加粗的状态，因为它被双星号所包含，"useful"被星号包含因此被显示成了斜体。

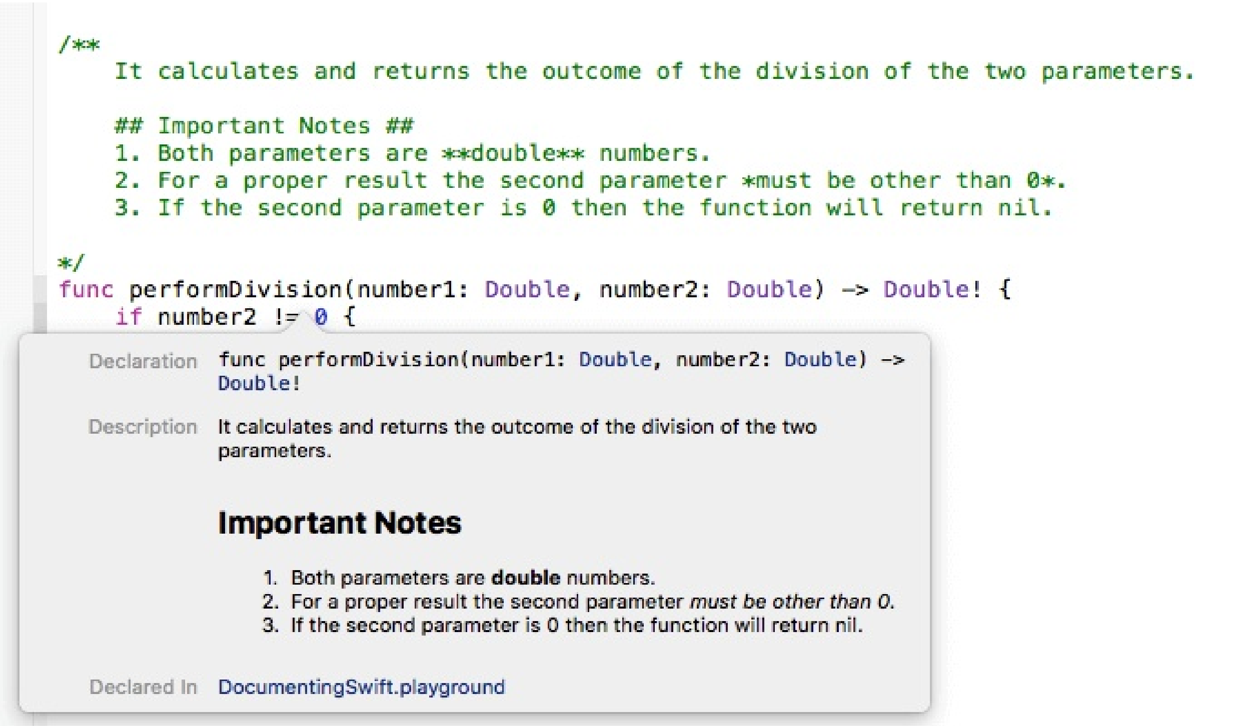
来看看另一个例子，这次使用一个函数:

```
/**
    It calculates and returns the outcome of the division of the two parameters.

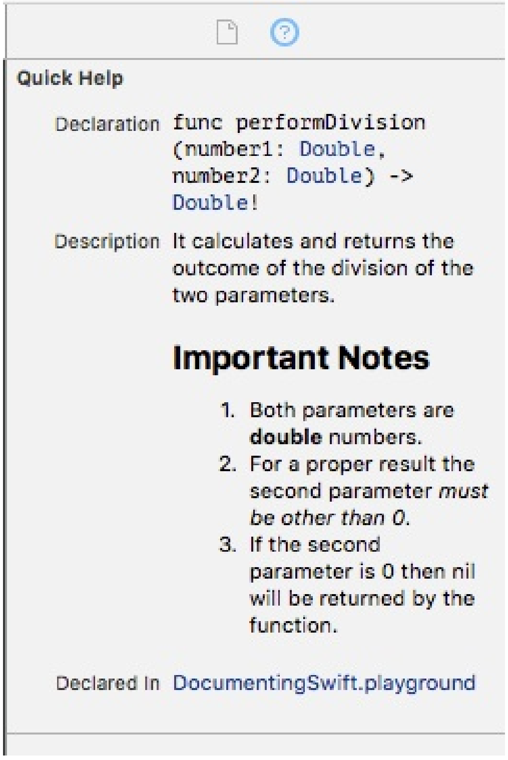
    ## Important Notes ##
    1. Both parameters are **double** numbers.
    2. For a proper result the second parameter *must be other than 0*.
    3. If the second parameter is 0 then the function will return nil.

*/
func performDivision(number1: Double, number2: Double) -> Double! {
    if number2 != 0 {
        return number1 / number2
    }
    else {
        return nil
    }
}
```

把上述代码复制粘贴进playground中，按住Option点击函数名的话，会看到显示出的快速帮助:



这里使用了两个新的Markdown元素，header与有序列表，在有序列表中的列表项应用了加粗与斜体格式。注意到使用Markdown的特殊字符来丰富文档中的文本样式是很轻松的一件事。在Quick Help Inspector中如下所示:



接下来，要在函数的文档中添加一个代码块。“`”符号除了可以创建代码块外，也常常用于在行内标记函数名

```
/**
    It doubles the value given as a parameter.

    ### Usage Example: ###
    `
```

```
let single = 5
let double = doubleValue(single)
print(double)
````

* Use the `doubleValue(_:)` function to get the double value of any number.
* Only ***Int*** properties are allowed.
```

*/ func doubleValue(value: Int) -> Int { return value* 2 }
```

~~~~~
效果如下：

![](http://www.appcoda.com/wp-content/uploads/2016/05/t52_4_quickhelp3.png)

最后让我们来给枚举添加一些文档，然后在另一个函数的函数体中使用它。在这里有意思的是枚举中的每一个元素都
有对应的文档：
```swift
/**
    My own alignment options.

    `
~~~~~

~~~~~
case Left
case Center
case Right
````
~~~~~

*/ enum AlignmentOptions { /// It aligns the text on the Left side. case Left

```
/// It aligns the text on the Center.
case Center

/// It aligns the text on the Right side.
case Right
```

}

func doSomething() { var alignmentOption: AlignmentOptions!

```
alignmentOption = AlignmentOptions.Left
```

}

````
现在，每当使用枚举中的元素时Xcode就会显示对应元素的文档：

![](http://www.appcoda.com/wp-content/uploads/2016/05/t52_5_quickhelp4.png)

## 使用关键字

使用Markdown语句只是文档化代码的其中一个方面，显然丰富文本的格式是很棒的，可以产生一个不错的文档，不过如果只干到这儿的话会落下一些东西的，来看看文档化代码的另一个方面——使用关键字。

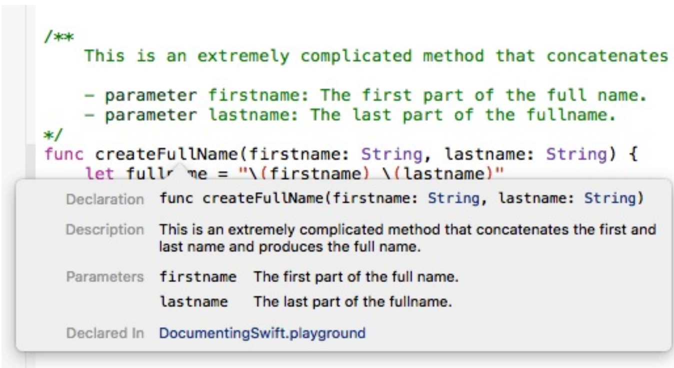
每当使用一个关键字时Xcode会自动识别它，用一个默认的格式去渲染文档(用第三方库时同理)。在标注文档中通用的代码结构来区分它们时使用关键字是很有效的。比如说，有用于高亮方法参数的、返回值的、类作者的、函数版本的关键字，还有很多，不用全部都使用，有些是很不常用的。熟悉一些常用的关键字，其他的可以用的时候再去查阅。

说了这么多，是时候来通过一些例子来练习一下关键字的使用了，第一个关键字用来标注一个方法或函数接受的参数：

```swift
/**
    This is an extremely complicated method that concatenates the first and last name and produces the full name.

    - Parameter firstname: The first part of the full name.
    - Parameter lastname: The last part of the fullname.
*/
func createFullName(firstname: String, lastname: String) {
    let fullname = "\(firstname) \(lastname)"
    print(fullname)
}
````

效果如下:



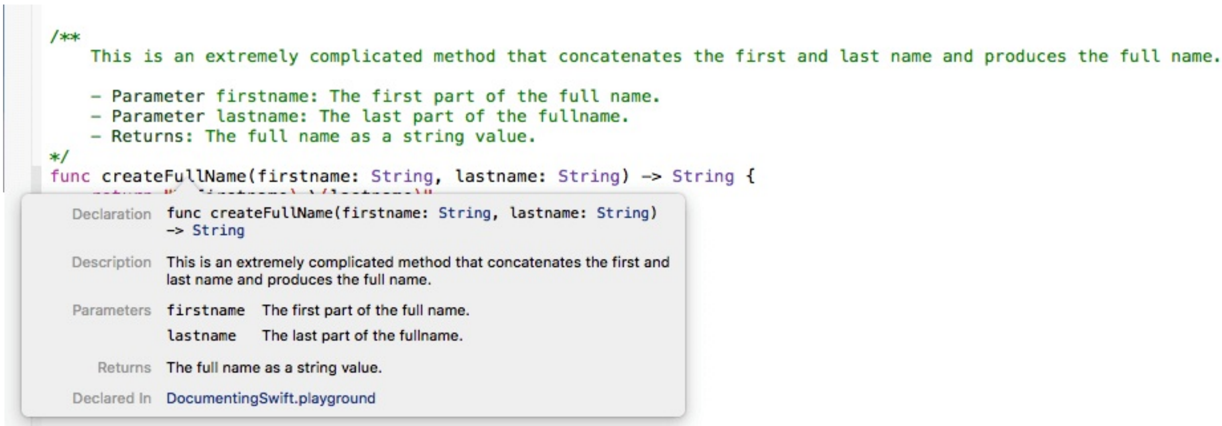
注意关键字前的破折号(-)，还有它们之间的空格，之后的内容则是实际的参数与其描述。在这里存在多少个参数就应该描述多少个参数(与合适的描述)。

来稍微修改一下上述的函数，让其返回fullname而不是仅仅打印它。这时，需要添加一个描述函数返回值的新关键字:
````

```
/**
    This is an extremely complicated method that concatenates the first and last name
and produces the full name.

     - Parameter firstname: The first part of the full name.
     - Parameter lastname: The last part of the fullname.
     - Returns: The full name as a string value.
*/
func createFullName(firstname: String, lastname: String) -> String {
    return "\(firstname) \(lastname)"
}
```

效果如下:



上面的两个关键字(parameter和returns)是最常用的。接着来看一个函数，执行与上述函数相逆的过程，将一个fullname拆分为firstname和lastname:

```
/**
    Another complicated function.

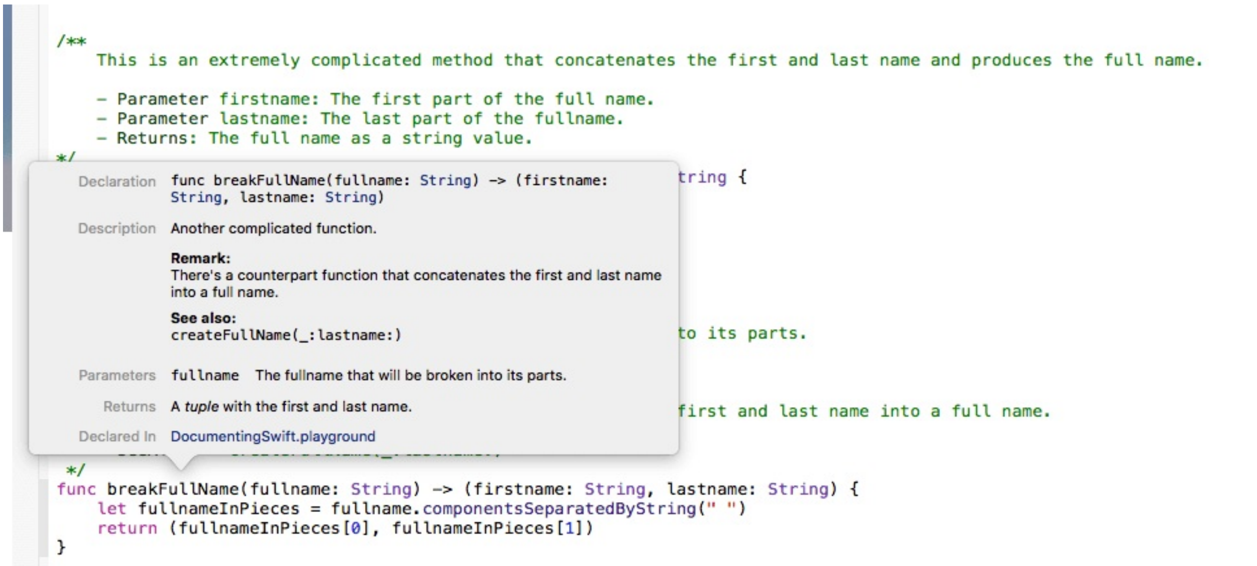
     - Parameter fullname: The fullname that will be broken into its parts.
     - Returns: A *tuple* with the first and last name.

     - Remark:
         There's a counterpart function that concatenates the first and last name into
a full name.

     - SeeAlso:  `createFullName(_:lastname:)`

*/
func breakFullName(fullname: String) -> (firstname: String, lastname: String) {
    let fullnameInPieces = fullname.componentsSeparatedByString(" ")
    return (fullnameInPieces[0], fullnameInPieces[1])
}
```

上面出现了两个新元素：Remark和SeeAlso关键字. 使用第一个关键字你可以高亮任何重要或特殊的点，来吸引阅读的人的注意。SeeAlso关键字常常用来添加与其他代码部分的关联(比如在这个例子中添加了之前一个函数的关联)，或提供一个真实URL的链接。在快速帮助中如下所示:



想象一下上述函数只是你分享给其他开发者的一部分，尽量将这个工作做得尽善尽美吧，你想要使用这个函数的人知道两个事实(或要求)：fullname参数不能为nil，fisrtname与lastname需要包含在fullname中，由空格分隔开，这样函数才能正常工作。你可以使用另外两个关键字：Precondition与Requires，将文档更新一下，如下所示:

```
/**
    Another complicated function.

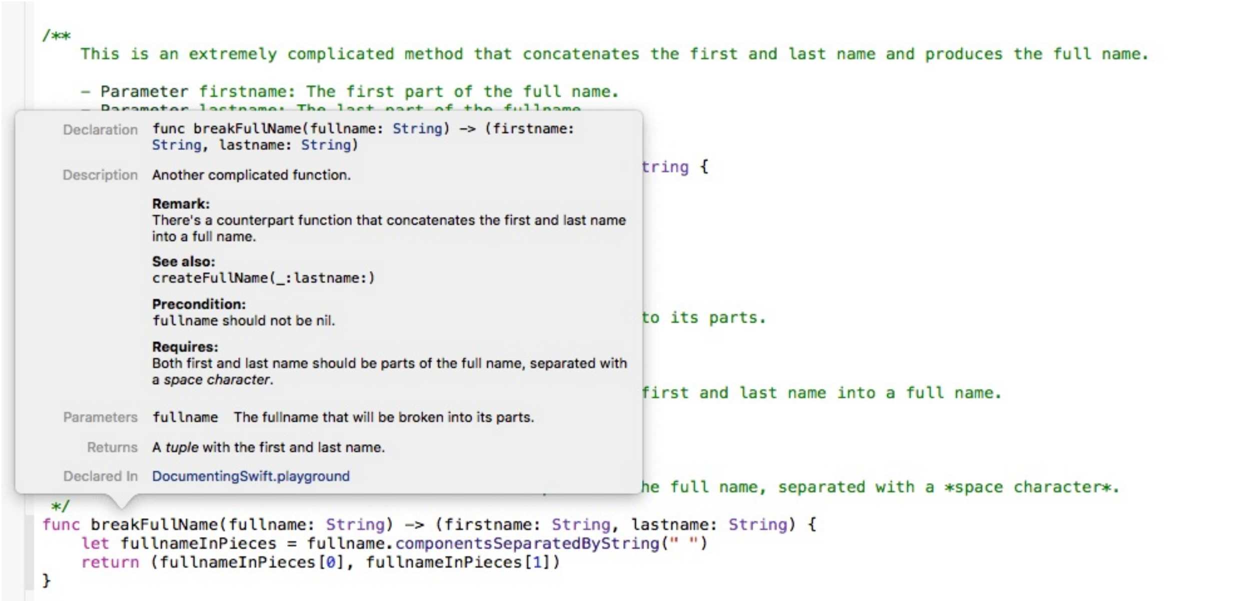
    - Parameter fullname: The fullname that will be broken into its parts.
    - Returns: A *tuple* with the first and last name.

    - Remark:
        There's a counterpart function that concatenates the first and last name into
a full name.

    - SeeAlso:  `createFullName(_:lastname:)`

    - Precondition: `fullname` should not be nil.
    - Requires: Both first and last name should be parts of the full name, separated w
ith a *space character*.

*/
func breakFullName(fullname: String) -> (firstname: String, lastname: String) {
    let fullnameInPieces = fullname.componentsSeparatedByString(" ")
    return (fullnameInPieces[0], fullnameInPieces[1])
}
```



着眼未来，你也许会记录一下以后要实现的功能:

```
- Todo: Support middle name in the next version.
```

也可以添加警告、版本、作者与提醒:

```
/**
    Another complicated function.

    - Parameter fullname: The fullname that will be broken into its parts.
    - Returns: A *tuple* with the first and last name.

    - Remark:
        There's a counterpart function that concatenates the first and last name into
a full name.

    - SeeAlso:  `createFullName(_:lastname:)`

    - Precondition: `fullname` should not be nil.
    - Requires: Both first and last name should be parts of the full name, separated w
ith a *space character*.

    - Todo: Support middle name in the next version.

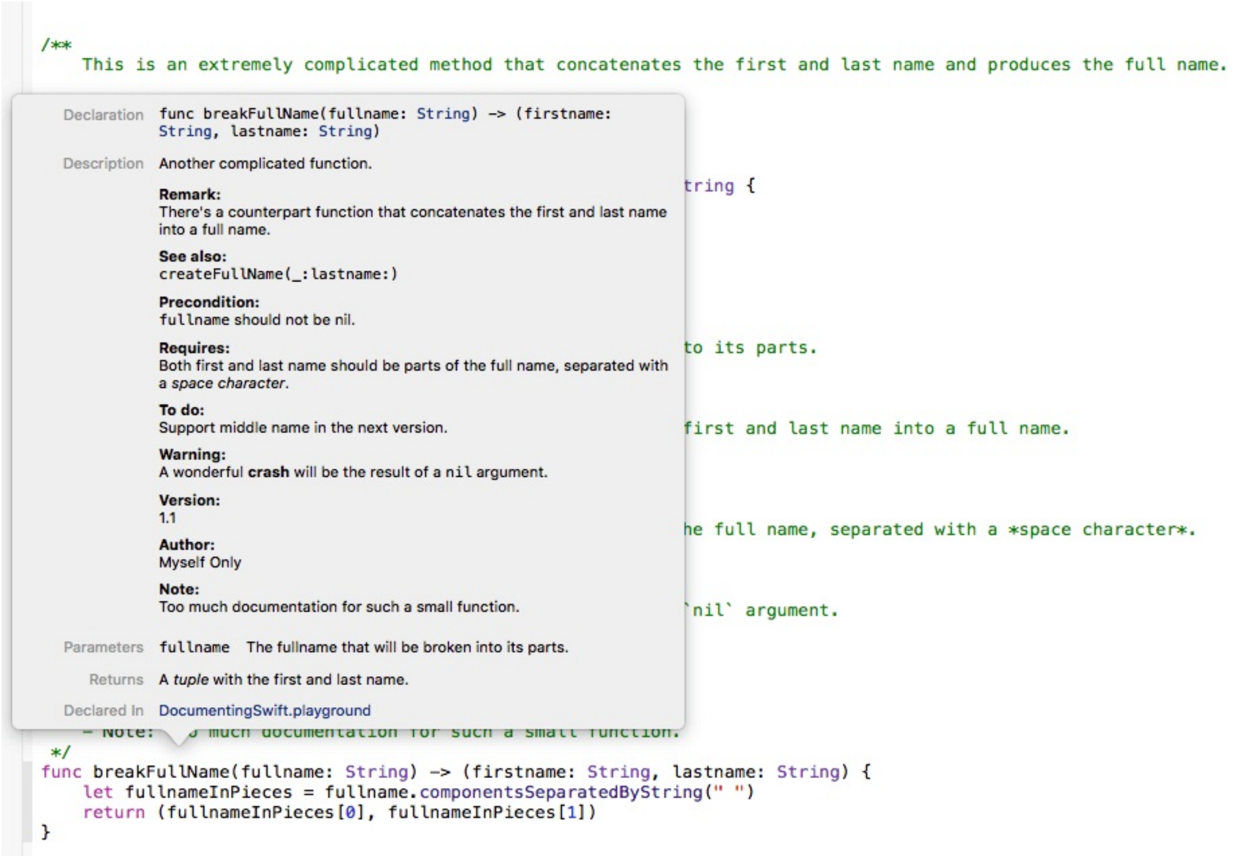
    - Warning: A wonderful **crash** will be the result of a `nil` argument.

    - Version: 1.1

    - Author: Myself Only

    - Note: Too much documentation for such a small function.
 */
func breakFullName(fullname: String) -> (firstname: String, lastname: String) {
    let fullnameInPieces = fullname.componentsSeparatedByString(" ")
    return (fullnameInPieces[0], fullnameInPieces[1])
}
```

这是快速帮助中的显示:



上述的细节理解程度取决于你的接受能力，对于重要的代码应该像前面代码片段中展示的那样来撰写文档，其他不太重要的部分使用基本的元素即可。

苹果提供了一个包含所有可用关键字的文档，可以在这里看到比较详细的介绍，不要错过哟。

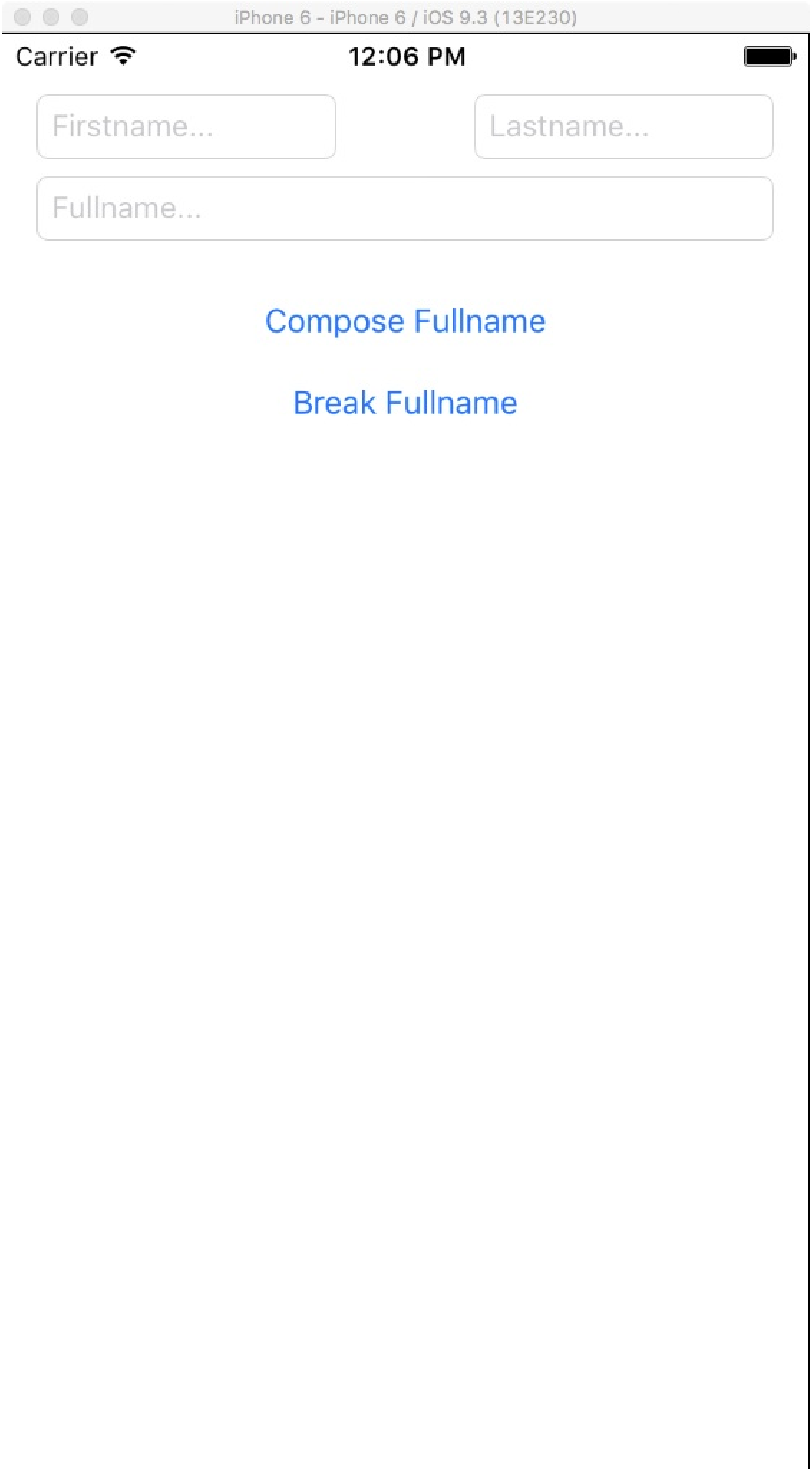
使用Jazzy生成文档页面

Jazzy是个用来为你的代码(不管是swift还是oc)生成苹果风格文档的一个好工具，实际上Jazzy创建了一个包含所有代码文档的独立页面。虽然它是个命令行工具，不过挺易用的。

下面来分析一下Jazzy的具体实现细节，可以看看GitHub上的介绍，也能看到它的环境要求与安装方法，只需要如下操作即可开始使用:

1. 打开你的终端。
2. 输入: [sudo] gem install jazzy.
3. 输入密码
4. 等待...

为了方便测试Jazzy，我准备了一个小型app，可以在这里下载，包含之前例子中的函数-组合与拆分fullname。


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
12:06 PM
Firstname...
Lastname...
Fullname...
Compose Fullname
Break Fullname

假设你已经下了app，来看看Jazzy怎么用。首先在终端里使用cd命令进入工程文件夹:

```
cd path_to_project_folder
```

最简单的用法是只需输入Jazzy，然后等待它执行完成，不过这些不会包含代码中没被标记为public的类或其他结构。因此如果你想包含所有的结构，需要输入如下命令:

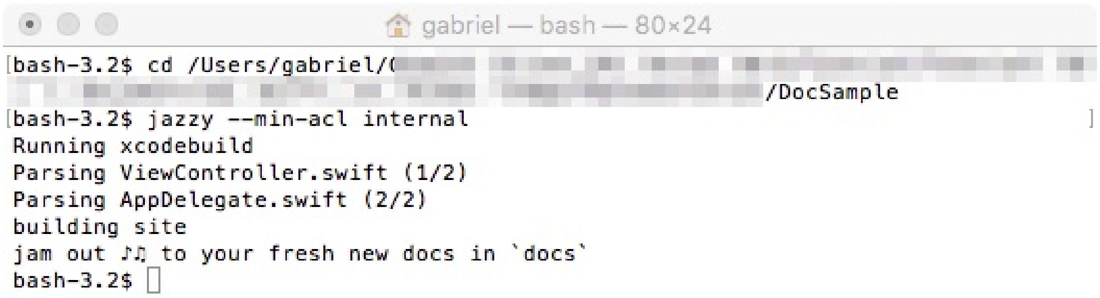
```
jazzy --min-acl internal
```

另外，如果你的swift不是最新版的则看不到Jazzy的输出，这样的话需要添加一个参数来指定xcode所支持的swift版本:

```
jazzy --swift-version 2.1.1 --min-acl internal
```

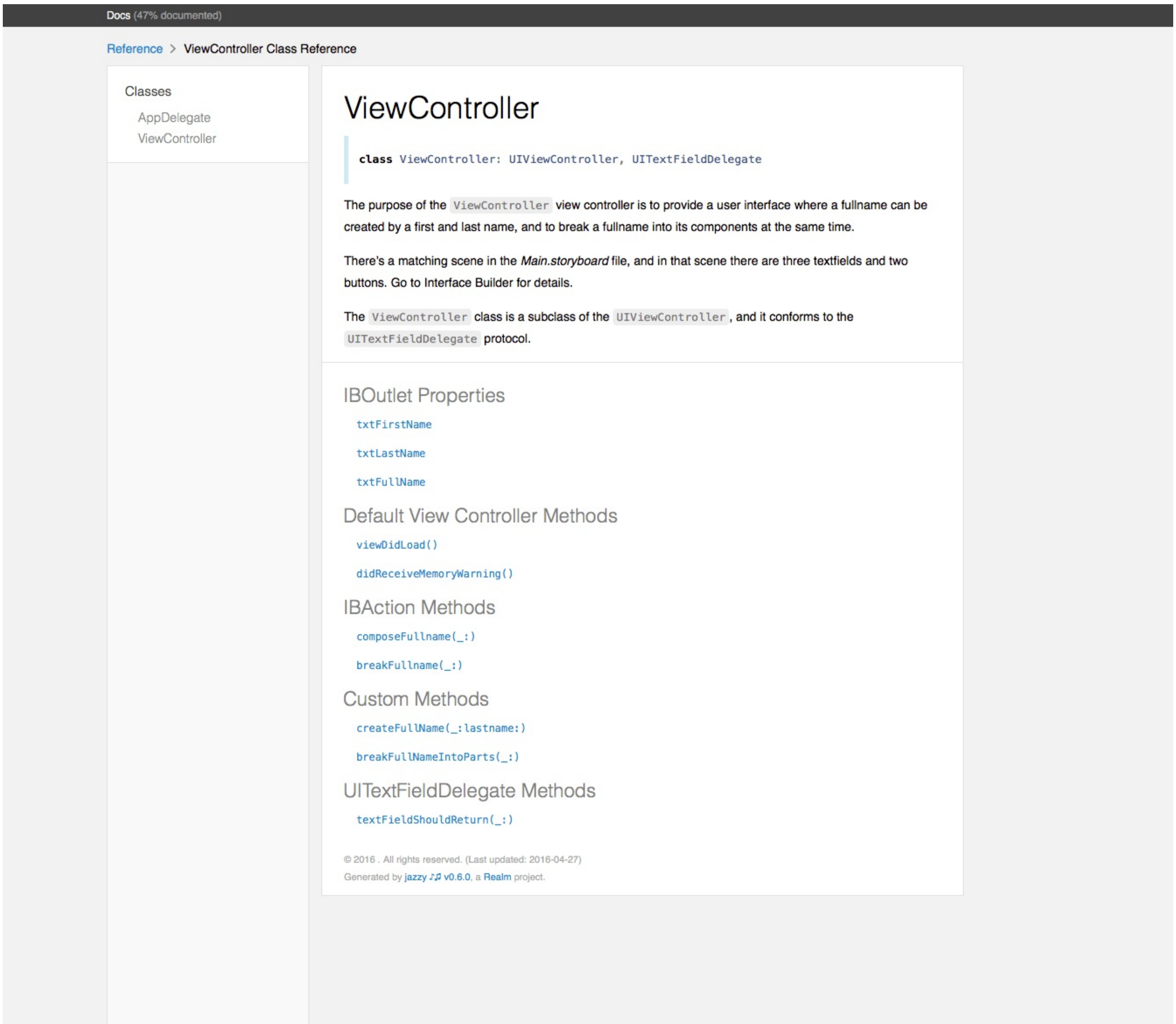
强烈建议你输入jazzy --help来看看Jazzy中所有可用的参数，也许会发现很多有用的东西可以按照自己的喜好来得到想要的结果。

执行生成文档页面，完成后的效果如下所示：



默认输出文件夹命名docs，保存在工程的根目录下(可以修改)。

使用Finder查看文件夹，用浏览器打开index.html，会看到默认的页面风格很像苹果文档的风格，点击链接试试看文档如何显示。好了，试试在你的工程中使用Jazzy吧。


Docs (47% documented)
Reference > ViewController Class Reference
Classes
AppDelegate
ViewController
ViewController
class ViewController: UIViewController, UITextFieldDelegate
The purpose of the ViewController view controller is to provide a user interface where a fullname can be created by a first and last name, and to break a fullname into its components at the same time.
There's a matching scene in the Main.storyboard file, and in that scene there are three textfields and two buttons. Go to Interface Builder for details.
The ViewController class is a subclass of the UIViewController, and it conforms to the UITextFieldDelegate protocol.
IBOutlet Properties
txtFirstName
txtLastName
txtFullName
Default View Controller Methods
viewDidLoad()
didReceiveMemoryWarning()
IBAction Methods
composeFullname(_:)
breakFullname(_:)
Custom Methods
createFullName(_:lastname:)
breakFullNameIntoParts(_:)
UITextFieldDelegate Methods
textFieldShouldReturn(_:)
© 2016 . All rights reserved. (Last updated: 2016-04-27)
Generated by jazzy ♪♫ v0.6.0, a Realm project.

Creating Gradient Colors Using CAGradientLayer

使用CAGradientLayer创建渐变色

> - 原文链接：Creating Gradient Colors Using CAGradientLayer
> - 原文作者：GABRIEL THEODOROPOULOS
> - 译者：yrq110

每个开发者在开发app时都会使用图像的色彩融合来得到一个赏心悦目的UI效果，尝试达到最佳的用户体验。然而有时需要做一些额外的工作，普通的颜色可能不会得到较好的展示效果，而使用渐变色则会产生更好的效果。我有一些使用渐变色的经验以及关于它的建议，在这里跟大家分享下。创建渐变色是很简单的因此开发者不去用的话是较可惜的。

那么，如何可以快速轻松地创建一个渐变效果？这里有三种方法。第一种，也是最不推荐的一种，使用具有渐变效果的图像，最大的缺陷就是如果没有设计出一系列不同状态的渐变的话就不能在运行过程中随意修改渐变效果。第二种方法是使用Core Graphics技术，需要掌握相关的知识(比如图形上下文、色彩空间等等)，并且Core Graphics框架是给相对高级的用户准备的，新人开发者可能用起来比较费劲。最后一种，也是最快速最简便的一种方式:使用CAGradientLayer对象。

CAGradientLayer是CALayer类的子类，顾名思义，用来产生渐变效果的类。使用4行代码、少于1分钟就可以生成一个简单的渐变效果，并且通过提供的少量属性的调整就能得到一个很好的结果。几乎所有种类渐变都有动画效果，因此不需要很费劲就能得到较好的结果。在下面的内容中会讨论其中的细节，很重要的一点是：你需要了解当你使用CAGradientLayer类来创建渐变效果时，实际上操作的是运用渐变的视图层。CAGradientLayer的一个缺点就是不支持径向渐变(radial gradient)，不过如果你主要使用的是线性渐变就完全满足需求了。

在之后的章节会介绍跟配置渐变效果有关的每个属性的详细信息，我会使用非常棒的色彩来演示，为了方便起见仅用两种颜色，对于demo来说足够了，你也可以使用更多的色彩。

目录

创建一个渐变层

创建一个渐变色彩层是项快速并且简单的任务，必须要制定一些参数。实际上你需要先设置少量的属性后就可以接着进行后面的操作，最花时间的也许是在操作渐变层时对最终结果参数的微调，对层属性设定不同的值会产生完全不同的效果。

让我们来逐步看看其中的细节，首先需要一个演示的app。打开Xcode创建一个新应用，确保选择的是单视图界面模板，跟着向导完成创建。

假设你已经准备好了一个新工程，点击ViewController.swift文件，在类中声明如下属性:

```
var gradientLayer: CAGradientLayer!
```

在之后会进行gradientLayer属性的设置，为了在目标视图中显示一个渐变层的最小实现步骤如下所示:

1. 初始化一个CAGradientLayer对象(这里是gradientLayer)。
2. 设置渐变层的尺寸。
3. 设置向导产生渐变效果的颜色。
4. 将渐变层作为子层添加到目标视图层。

除了上面这些步骤，还有一些需要配置的参数，之后会讲到。现在先主要关注这些步骤即可，简便起见，使用ViewController类中的默认视图作为目标视图，使用渐变色彩来填充它。

来操作一下ViewController类，创建一个初始化与设置渐变层属性默认值的新方法:

```
func createGradientLayer() {
    gradientLayer = CAGradientLayer()

    gradientLayer.frame = self.view.bounds

    gradientLayer.colors = [UIColor.redColor().CGColor, UIColor.yellowColor().CGColor]

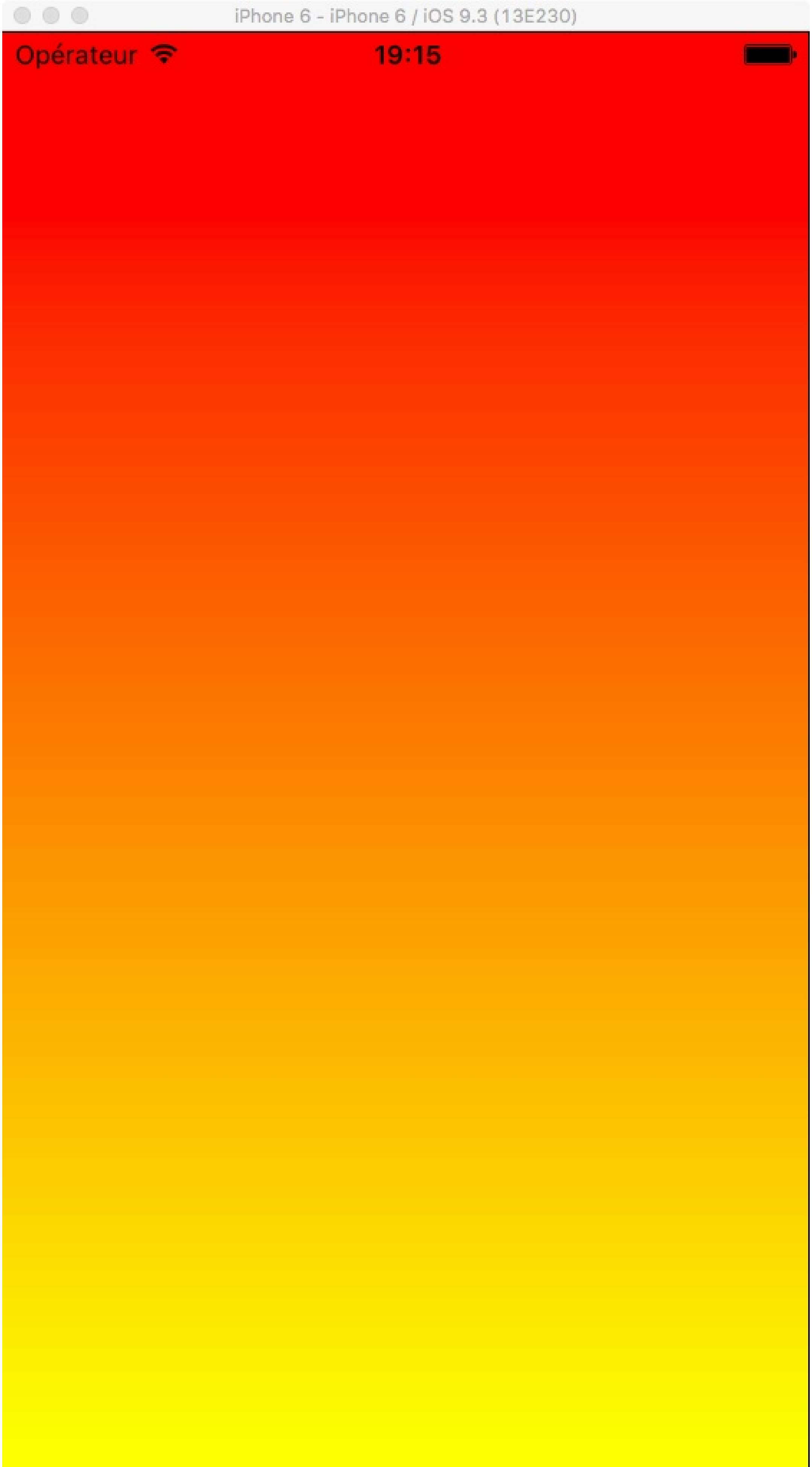
    self.view.layer.addSublayer(gradientLayer)
}
```

首先初始化了渐变层对象，设置它的尺寸，使其与视图控制器视图的边界属性相等。接着指定渐变效果的色彩，最后将渐变层作为子层添加到默认的视图层上。

调用上述的viewWillAppear(_:)方法:

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    createGradientLayer()
}
```

…然后运行app，效果如下:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Opérateur
19:15

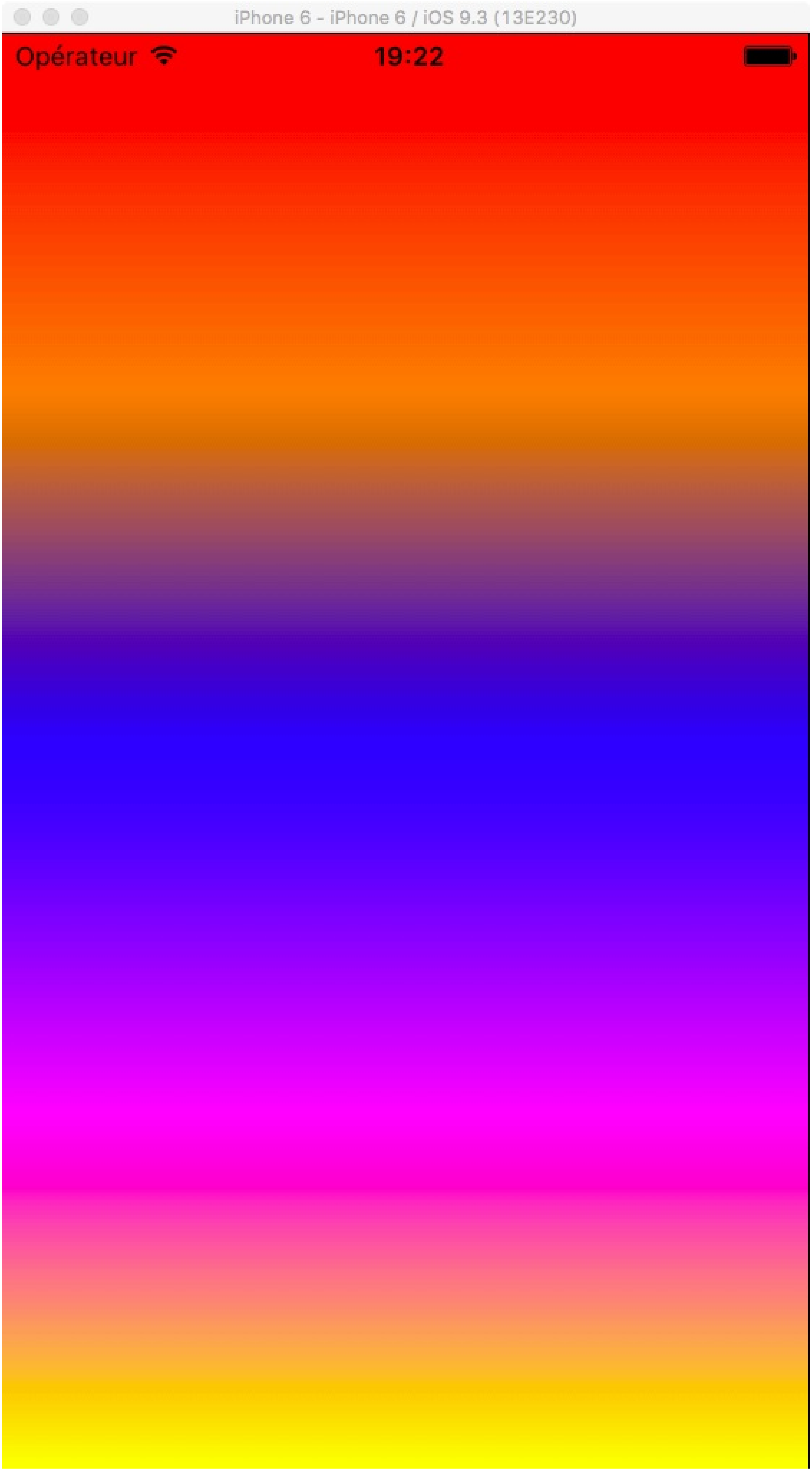
不是很糟，仅用了4行简单的代码，来深入一下其中的细节。

渐变色

虽然上面的代码很简单，不过其中包含一个重要的属性:色彩属性。首先我觉得我不需要解释:不设置色彩的话就不会有一丝渐变效果。其次，这个属性需要接受一个色彩数组，而不是一个UIColor对象。在上面的例子中只用了两种颜色，你可以用更多的颜色。比如说设置如下的色彩集:

```
gradientLayer.colors = [UIColor.redColor().CGColor, UIColor.orangeColor().CGColor, UIColor.blueColor().CGColor, UIColor.magentaColor().CGColor, UIColor.yellowColor().CGColor]
```

...运行app后如下所示:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Opérateur
19:22

色彩属性有个很棒的特性就是可以设置动画，意味着你可以使用动画转换的方式变换渐变效果，为了证明它，来创建一个可用的由色彩数组组成的集合，点击视图时切换执行色彩的色彩集，不过色彩集之间的变换是具有动画效果的。

首先，在ViewController类的顶部，在gradientLayer属性之后声明两种新属性:

```
var colorSets = [[CGColor]]()

var currentColorSet: Int!
```

colorSets数组接受的元素是由CGColor对象组成的数组，currentColorSet表示当前执行渐变效果的色彩集。

现在来创建色彩集，下面的颜色只是个例子，你可以使用想要的颜色N:

```
func createColorSets() {
    colorSets.append([UIColor.redColor().CGColor, UIColor.yellowColor().CGColor])
    colorSets.append([UIColor.greenColor().CGColor, UIColor.magentaColor().CGColor])
    colorSets.append([UIColor.grayColor().CGColor, UIColor.lightGrayColor().CGColor])

    currentColorSet = 0
}
```

上面的方法中，将由不同色彩组成的数组添加进了colorSets数组中，并且还设置了currentColorSet属性的初始值。

来在viewDidLoad()方法中调用它:

```
override func viewDidLoad() {
    super.viewDidLoad()

    createColorSets()
}
```

还需要稍微修改下这个createGradientLayer()方法:

```
gradientLayer.colors = [UIColor.redColor().CGColor, UIColor.orangeColor().CGColor, UIC
olor.blueColor().CGColor, UIColor.magentaColor().CGColor, UIColor.yellowColor().CGColor
]
```

...用下面的代码替换:

```
gradientLayer.colors = colorSets[currentColorSet]
```

这样就会使用通过currentColorSet属性所指定的色彩集，而不是之前使用的默认色彩。

之前我提到过：点击视图可以在不同色彩集间通过动画的方式变换，这意味着需要给视图添加一个点击的手势识别器，在viewDidLoad()中添加如下代码:

```
override func viewDidLoad() {
    ...

    let tapGestureRecognizer = UITapGestureRecognizer(target: self, action: #selector(
ViewController.handleTapGesture(_:)))
    self.view.addGestureRecognizer(tapGestureRecognizer)
}
```

若视图接收到点击手势时会调用handleTapGesture(_:)方法，离完成不远了。注意到下面代码中用到的CABasicAnimation，使用它来产生层色彩属性变换的动画效果，这里我认为你已经具备了有关CABasicAnimation的知识，了解动画是如何实现的，如果不是的话请去google下。没有包含某些属性，只使用了执行动画所需的基本属性。

```
func handleTapGesture(gestureRecognizer: UITapGestureRecognizer) {
    if currentColorSet < colorSets.count - 1 {
        currentColorSet! += 1
    }
    else {
        currentColorSet = 0
    }

    let colorChangeAnimation = CABasicAnimation(keyPath: "colors")
    colorChangeAnimation.duration = 2.0
    colorChangeAnimation.toValue = colorSets[currentColorSet]
    colorChangeAnimation.fillMode = kCAFillModeForwards
    colorChangeAnimation.removedOnCompletion = false
    gradientLayer.addAnimation(colorChangeAnimation, forKey: "colorChange")
}
```

首先我们要决定下一个色彩集的索引，如果当前选择的色彩集是colorSets数组中的最后一个，则从最开始(currentColorSet = 0)选取，否则给currentColorSet属性+1。

第二部分的代码是与动画相关的，duration与toValue是最重要的属性，duration意味着动画持续多久，toValue则是设置的期望目标色彩集。kCAFillModeForwards属性使动画结束后保持最后的状态，不会变为初始色彩，不过这并不是永久的，需要明确设置新的渐变色彩，何时设置？在动画完成后即可，使用下面这个方法，当CABasicAnimation完成时会调用它:

```
override func animationDidStop(anim: CAAnimation, finished flag: Bool) {
    if flag {
        gradientLayer.colors = colorSets[currentColorSet]
    }
}
```

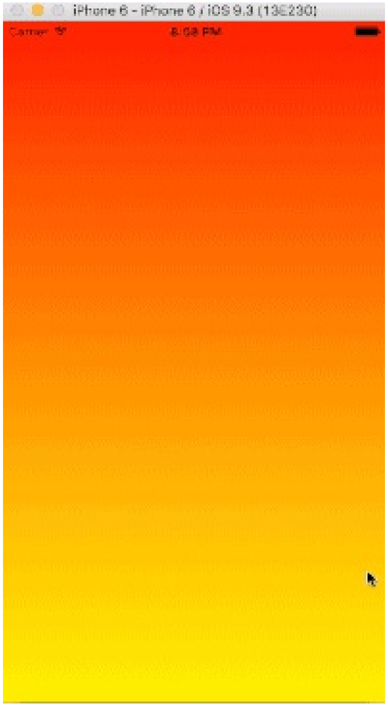
再修改下handleTapGesture(_:)方法，这样上面的方法就起作用了:

```
func handleTapGesture(gestureRecognizer: UITapGestureRecognizer) {
    ...

    // Add this line to make the ViewController class the delegate of the animation ob
ject.
    colorChangeAnimation.delegate = self

    gradientLayer.addAnimation(colorChangeAnimation, forKey: &quot;colorChange&quot;)
}
```

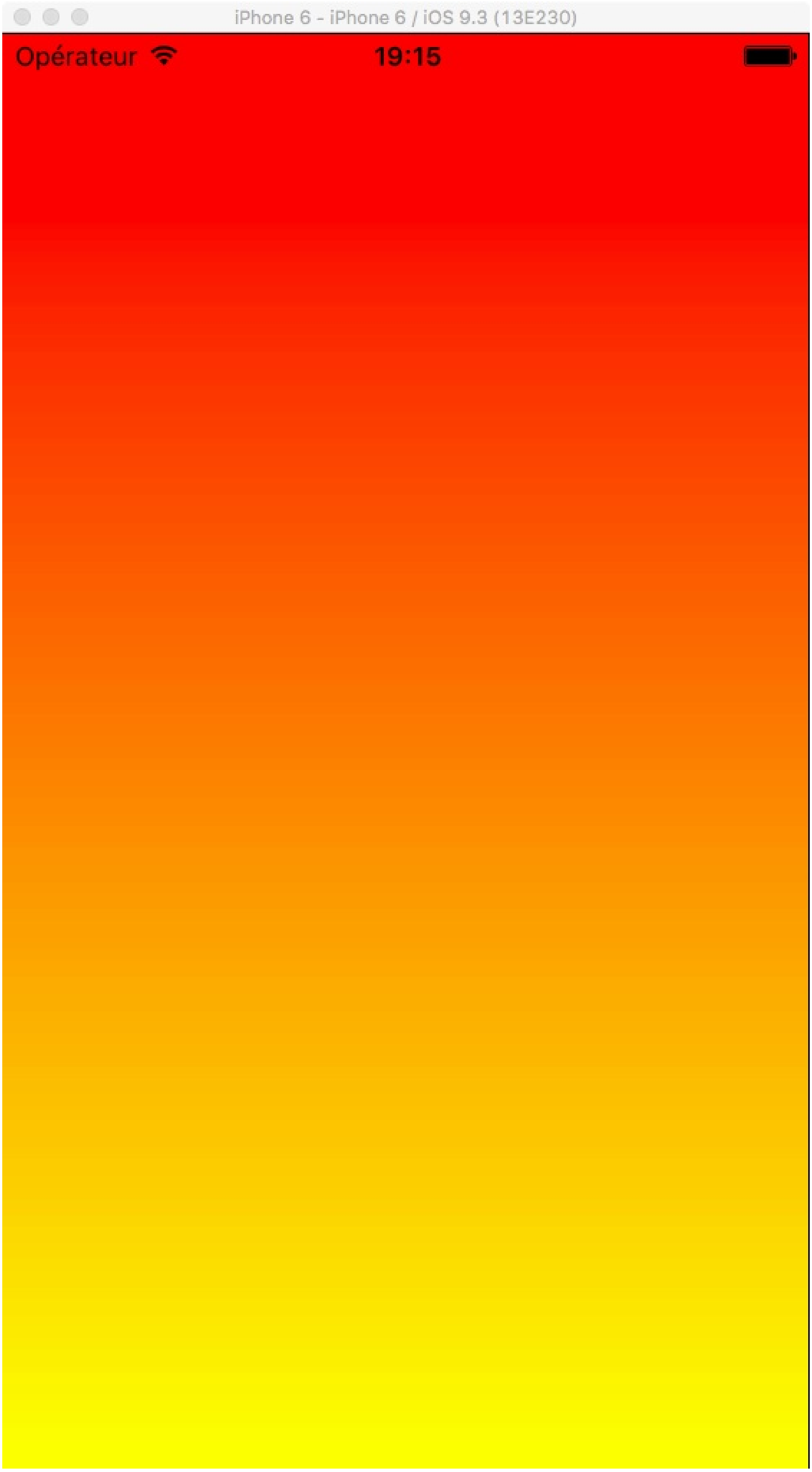
搞定了。可以自由的调整动画持续时间，我设成了2秒，可以看到渐变的效果很明显:



颜色位置

知道渐变效果中颜色的设置与改变是渐变层的基础，离最终的目标还是有一段距离的。还需要知道如何修改层中每个色彩所覆盖的区域，这也是很重要的。

如果你看看之前创建的那些渐变效果的话会发现默认情况下每个颜色都会占据层中的一半区域:

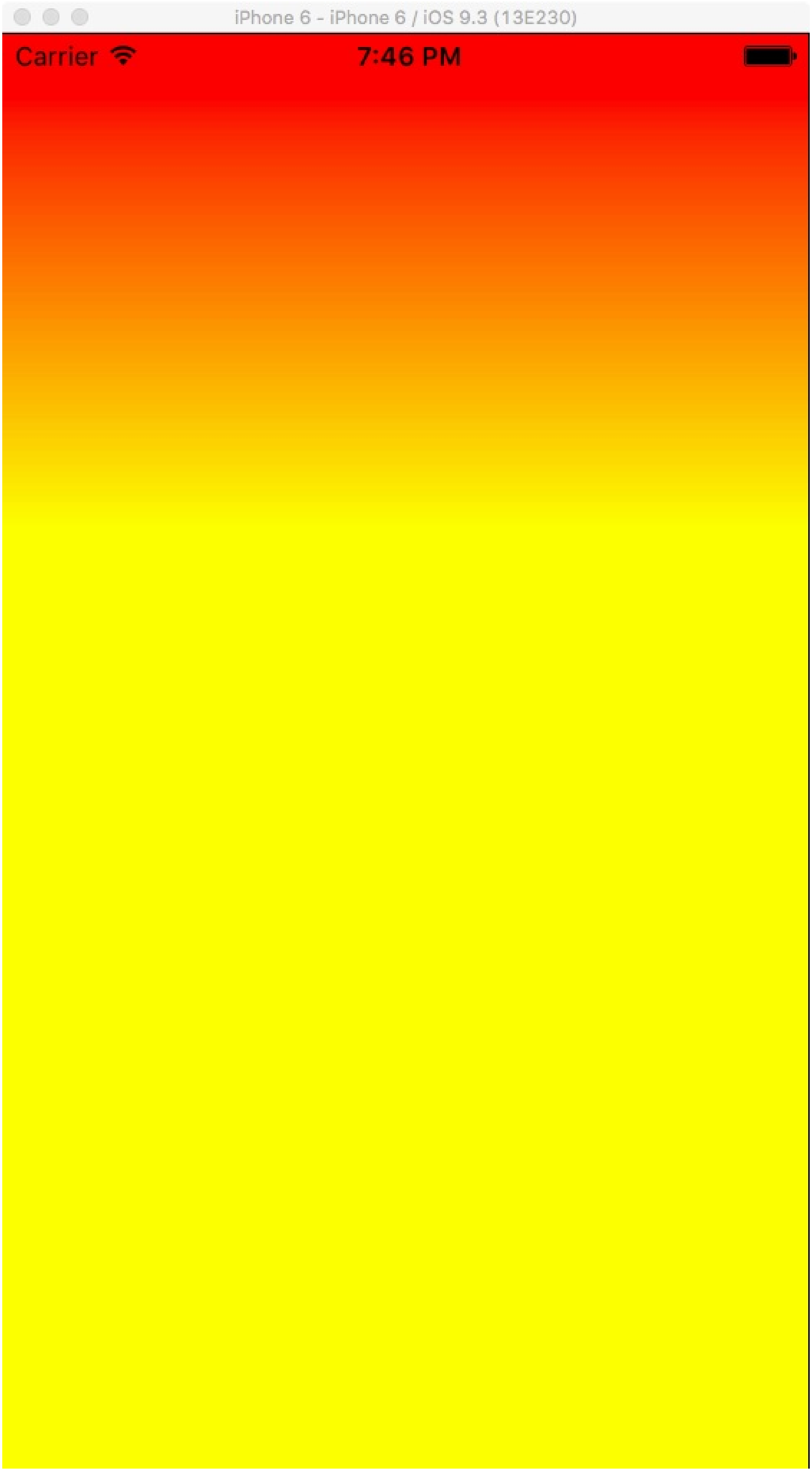

iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Opérateur
19:15

可以使用CAGradientLayer类中的locations属性来改变这个区域，locations属性接受一个由NSNumber对象组成的数组，每个数字都决定一个色彩的起始位置，并且还需要手动设置为0.0到1.0范围内的数。

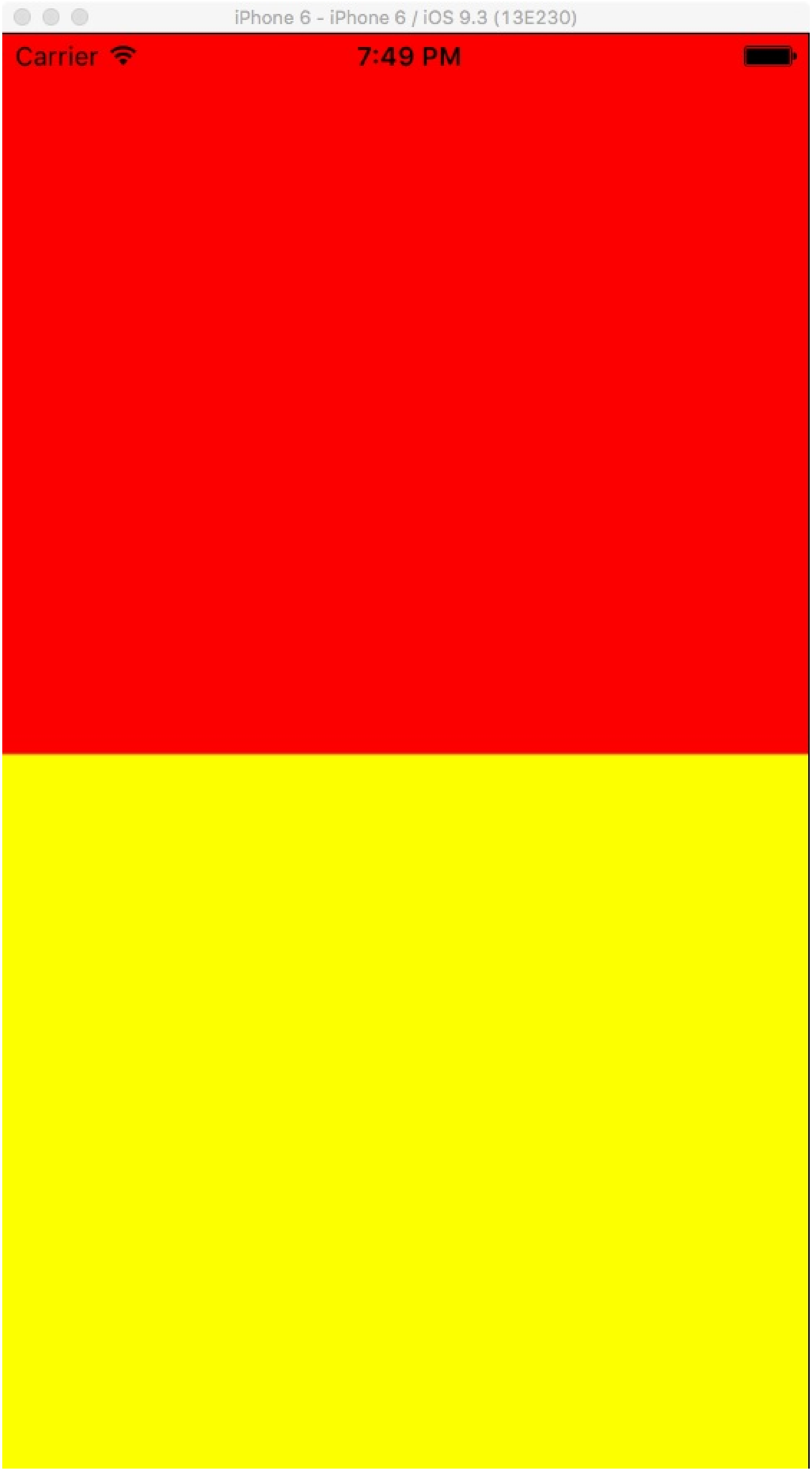
百闻不如一见，在createGradientLayer()方法中添加如下行:

```
gradientLayer.locations = [0.0, 0.35]
```

再次运行app，看看渐变效果的变化:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
7:46 PM

第二个颜色的起始位置是根据我们在locations数组中设置的第二个值而定的，相当于第二个颜色覆盖了层的65%(1.0 – 0.35 = 0.65)的区域。采取一项预防措施，确保每个颜色的位置不会比下一个颜色位置的值大，否则会发生的非期望的重叠现象:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
7:49 PM

若你想在app中看到上图的效果，只需要把色彩的location设置成[0.5, 0.35]即可。

来给demo再加点料吧，给视图添加一个点击手势识别器。这次将点击事件设为需要两个手指的操作，在viewDidLoad()中添加如下行:

```
override func viewDidLoad() {
    ...

    let twoFingerTapGestureRecognizer = UITapGestureRecognizer(target: self, action: #
selector(ViewController.handleTwoFingerTapGesture(_:)))
    twoFingerTapGestureRecognizer.numberOfTouchesRequired = 2
    self.view.addGestureRecognizer(twoFingerTapGestureRecognizer)
}
```

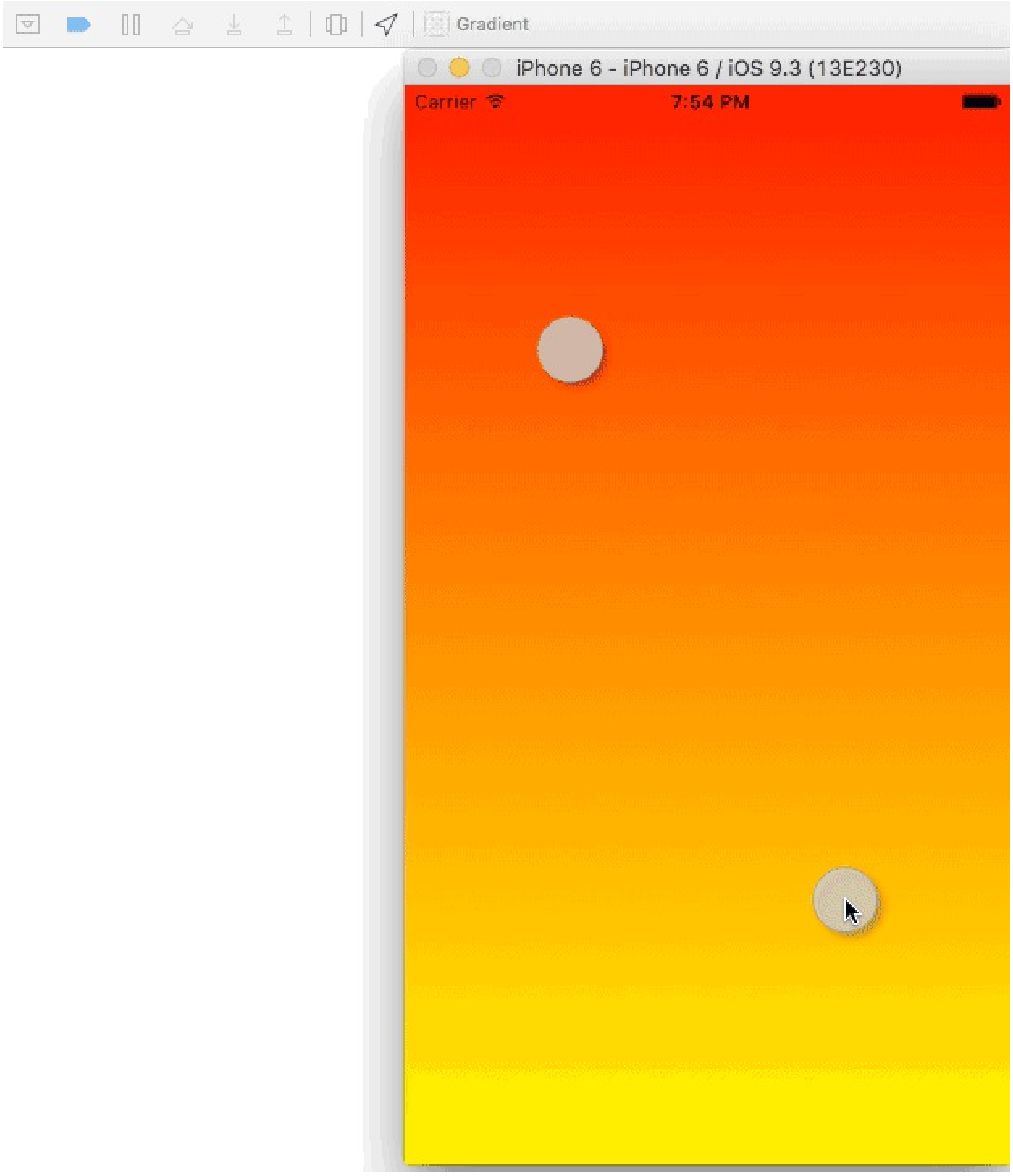
在handleTwoFingerTapGesture(_:)方法中会给不同色彩分配随机的位置，需要确保firstColorLocation总小于secondColorLocation，另外在改变位置时会将新的位置值实时打印在控制台中:

```
func handleTwoFingerTapGesture(gestureRecognizer: UITapGestureRecognizer) {
    let secondColorLocation = arc4random_uniform(100)
    let firstColorLocation = arc4random_uniform(secondColorLocation - 1)

    gradientLayer.locations = [NSNumber(double: Double(firstColorLocation)/100.0), NSN
umber(double: Double(secondColorLocation)/100.0)]

    print(gradientLayer.locations!)
}
```

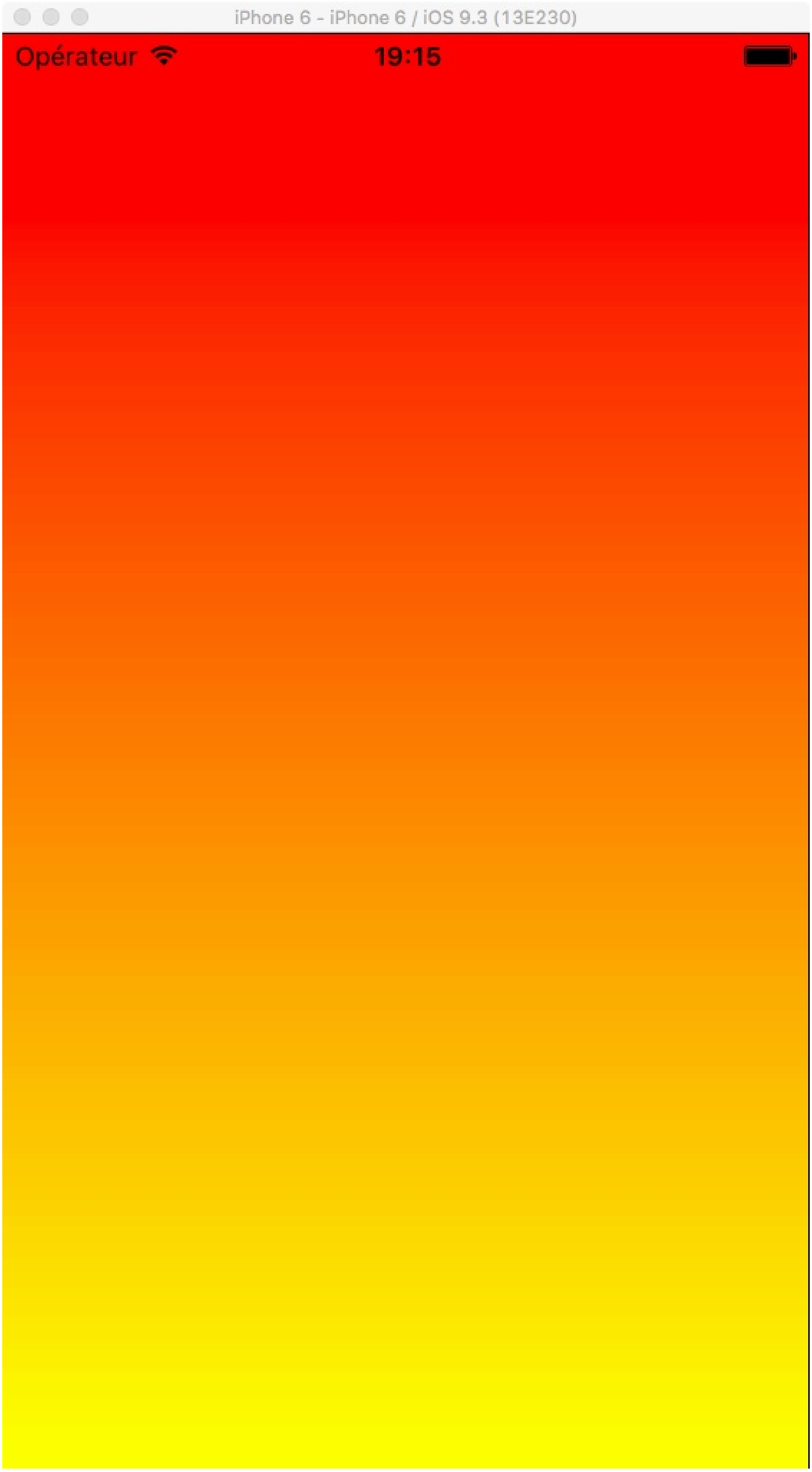
使用两个手指操作的效果如下:



注意默认情况下locations属性是空的，需要考虑到可能会崩溃的问题。虽然现在在demo中为了方便起见只使用了两种颜色，在使用更多颜色产生渐变效果时上面的方法同样适用。

渐变方向

已经知道了渐变层颜色属性的一些相关操作，来着手学习一下如何处理渐变效果的方向吧。首先看看下面这张图:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Opérateur
19:15

一眼就能看出渐变效果是从顶部开始朝向底部，实际上这是渐变层颜色的默认方向，跟其他属性一样，可以通过重写得到一个不同的方向。

CAGradientLayer类提供了两个指定渐变方向的属性:

- startPoint-起始点
- endPoint-终止点

一般用CGPoint值来表示上面的属性，x与y值的范围都应为0.0到1.0之间。起始点描述了第一个颜色的起始坐标，终止点描述了最后一个颜色的终止坐标，用这种方式来决定渐变的方向。这里有个重要的细节:坐标是与操作系统的坐标空间一致的。

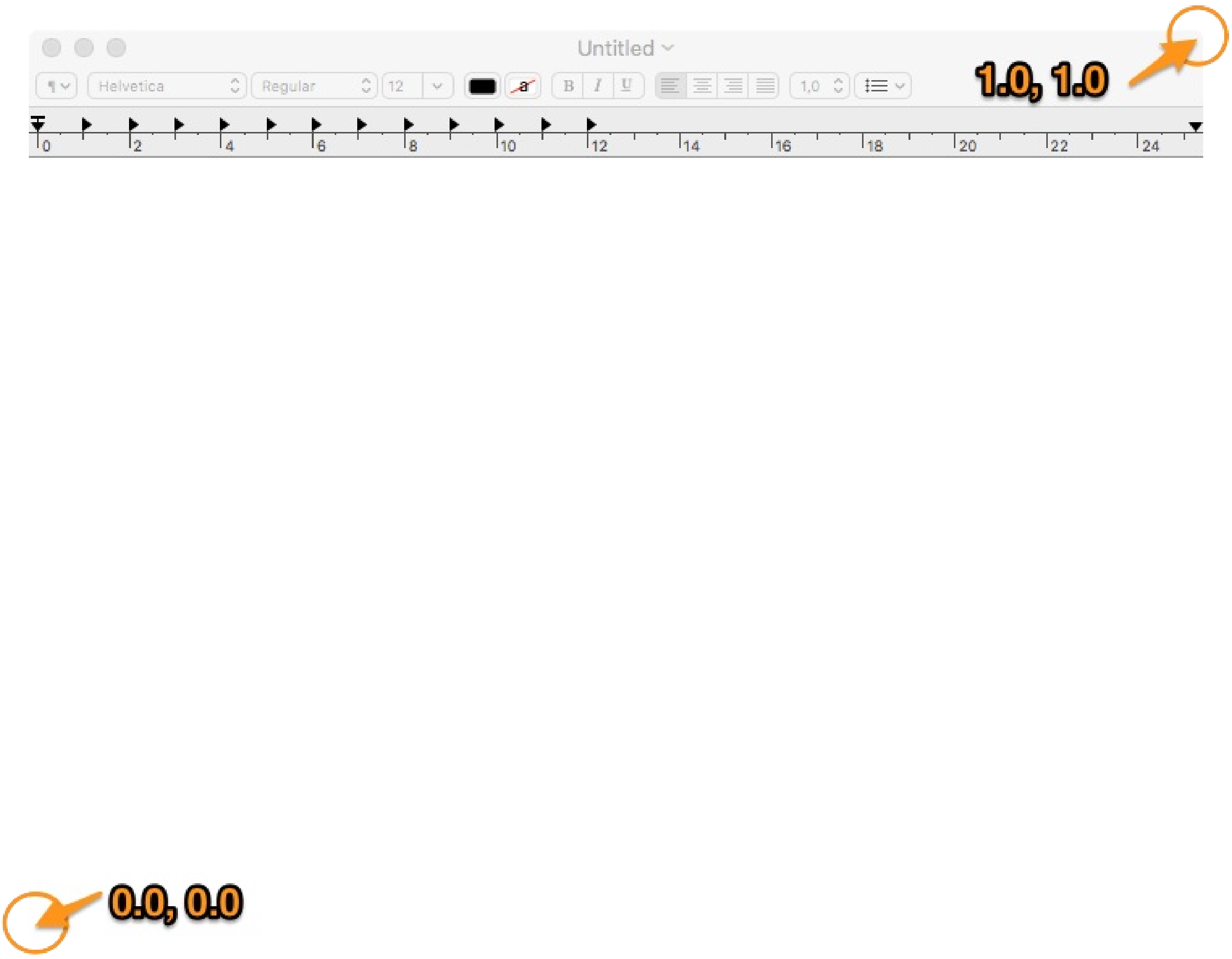
这意味着什么?

为了更好理解，看看下面这张图:


9:41 AM
100%
0.0, 0.0
1.0, 1.0

在iOS中，零点(起始点)是屏幕左上角的点(x = 0.0, y = 0.0)，终止点是右下角的点(x = 1.0, y = 1.0)，其它点都在这个坐标系中。

上面的坐标系与其他操作系统的并不相同，比如说Mac上的一个窗口(这里是个文本编辑器):



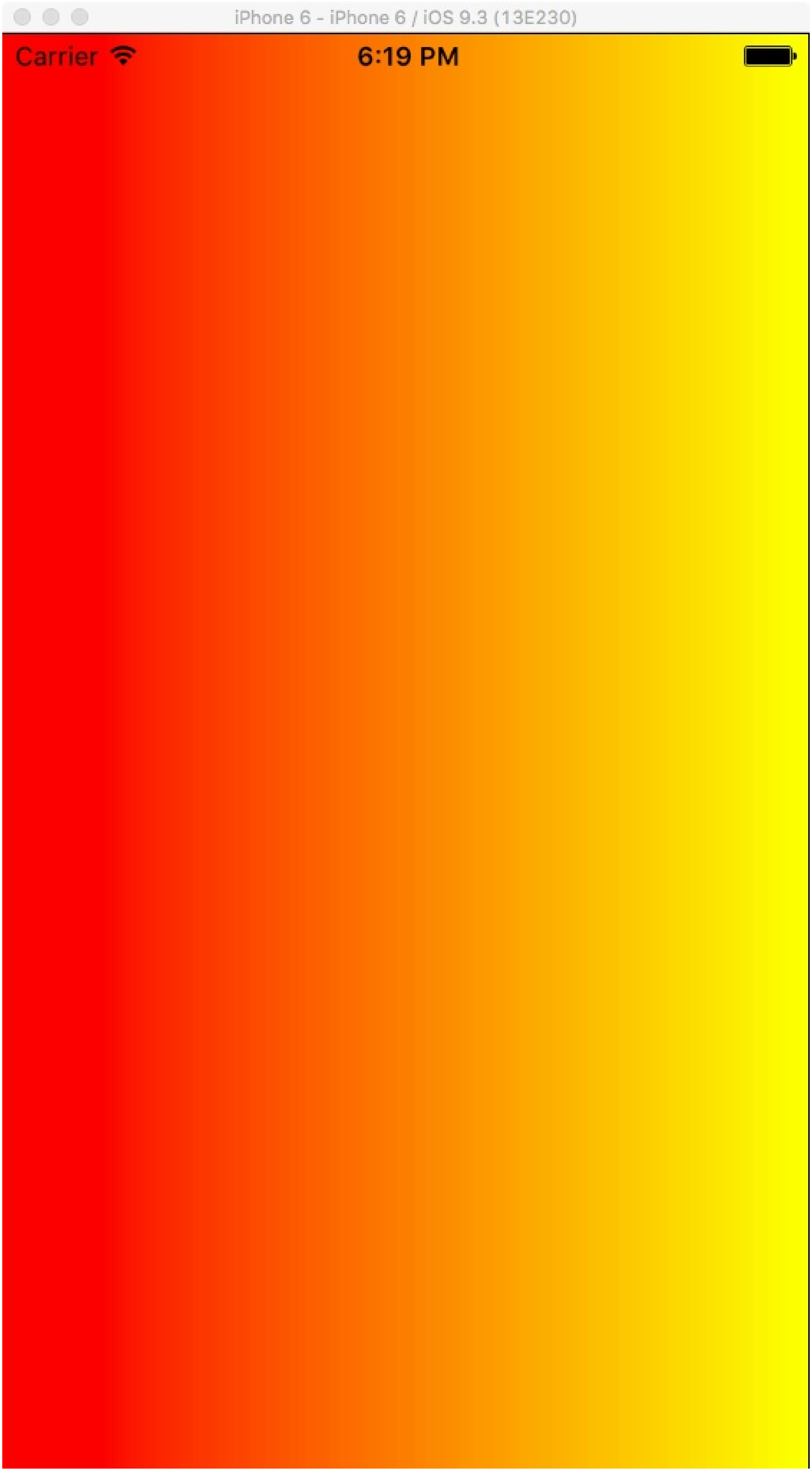
这里的起始点是在左下角，终止点是右上角，坐标系与iOS的是不同的。

默认的起始点是(0.5, 0.0)，终止点是(0.5, 1.0)，注意到x值没变，y值从0.0(顶部)变成了1.0(底部)，创建了向下的方向。来尝试一下不同的方向，在createGradientLayer()方法中添加如下两行:

```
func createGradientLayer() {
    ...

    gradientLayer.startPoint = CGPointMake(0.0, 0.5)
    gradientLayer.endPoint = CGPointMake(1.0, 0.5)
}
```

x值从0.0变为了1.0，y值不变，产生了一个向右的渐变效果:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
6:19 PM

为了更好的理解渐变方向，在0.0到1.0的范围内随意修改x和y的值看看改变后产生的效果。我们来给demo中添加一个新功能，使用拖动手势来改变方向，支持如下方向:

- 向上
- 向下
- 向右
- 向左
- 从左上到右下
- 从右上到左下
- 从左下到右上
- 从右下到左上

回到工程中，新建一个枚举来表示所有的方向:

```
enum PanDirections: Int {
    case Right
    case Left
    case Bottom
    case Top
    case TopLeftToBottomRight
    case TopRightToBottomLeft
    case BottomLeftToTopRight
    case BottomRightToTopLeft
}
```

之后再ViewController类中声明一个标示渐变方向的新属性:

```
var panDirection: PanDirections!
```

panDirection属性会获取手势移动的方向，下面需要处理一个分为两个步骤的任务:首先决定期望的方向分配给上面的属性，然后等拖动手势结束后设置对应的起始值与终止值。

还需要给视图添加一个拖动手势的识别器，在viewDidLoad()方法中添加如下代码:

```
override func viewDidLoad() {
    ...

    let panGestureRecognizer = UIPanGestureRecognizer(target: self, action: #selector(
ViewController.handlePanGestureRecognizer(_:)))
    self.view.addGestureRecognizer(panGestureRecognizer)
}
```

在实现handlePanGestureRecognizer(_:)方法时使用了手势识别器的速度(velocity)属性，若某一方向(x或y)上的速度大于300.0，则取这个方向。逻辑很好理解: 优先检测水平方向的速度，第二层中检测垂直方向的速度，代码如下:

```
func handlePanGestureRecognizer(gestureRecognizer: UIPanGestureRecognizer) {
    let velocity = gestureRecognizer.velocityInView(self.view)

    if gestureRecognizer.state == UIGestureRecognizerState.Changed {
        if velocity.x > 300.0 {
            // In this case the direction is generally towards Right.
            // Below are specific cases regarding the vertical movement of the gesture.


            if velocity.y > 300.0 {
                // Movement from Top-Left to Bottom-Right.
                panDirection = PanDirections.TopLeftToBottomRight
            }
            else if velocity.y < -300.0 {
                // Movement from Bottom-Left to Top-Right.
                panDirection = PanDirections.BottomLeftToTopRight
            }
            else {
                // Movement towards Right.
                panDirection = PanDirections.Right
            }
        }
        else if velocity.x < -300.0 {
            // In this case the direction is generally towards Left.
            // Below are specific cases regarding the vertical movement of the gesture.


            if velocity.y > 300.0 {
                // Movement from Top-Right to Bottom-Left.
                panDirection = PanDirections.TopRightToBottomLeft
            }
            else if velocity.y < -300.0 {
                // Movement from Bottom-Right to Top-Left.
                panDirection = PanDirections.BottomRightToTopLeft
            }
            else {
                // Movement towards Left.
                panDirection = PanDirections.Left
            }
        }
        else {
            // In this case the movement is mostly vertical (towards bottom or top).

            if velocity.y > 300.0 {
                // Movement towards Bottom.
                panDirection = PanDirections.Bottom
            }
            else if velocity.y < -300.0 {
```

```
                // Movement towards Top.
                panDirection = PanDirections.Top
            }
            else {
                // Do nothing.
                panDirection = nil
            }
        }
    }
    else if gestureRecognizer.state == UIGestureRecognizerState.Ended {
        changeGradientDirection()
    }
}
```

上述代码中需要注意两点(不仅仅是决定拖动手势的方向):

1. 若没有满足任何条件则panDirection为nil。
2. 方向由手势改变的状态所决定，手势结束时调用了changeGradientDirection()方法，从panDirection属性中得到新的方向。

下面是个简单的实现，仅设置了渐变层的一些起始点与终止点，观察一下x与y是如何随着手势方向变化的:

```
func changeGradientDirection() {
    if panDirection != nil {
        switch panDirection.rawValue {
        case PanDirections.Right.rawValue:
            gradientLayer.startPoint = CGPointMake(0.0, 0.5)
            gradientLayer.endPoint = CGPointMake(1.0, 0.5)

        case PanDirections.Left.rawValue:
            gradientLayer.startPoint = CGPointMake(1.0, 0.5)
            gradientLayer.endPoint = CGPointMake(0.0, 0.5)

        case PanDirections.Bottom.rawValue:
            gradientLayer.startPoint = CGPointMake(0.5, 0.0)
            gradientLayer.endPoint = CGPointMake(0.5, 1.0)

        case PanDirections.Top.rawValue:
            gradientLayer.startPoint = CGPointMake(0.5, 1.0)
            gradientLayer.endPoint = CGPointMake(0.5, 0.0)

        case PanDirections.TopLeftToBottomRight.rawValue:
            gradientLayer.startPoint = CGPointMake(0.0, 0.0)
            gradientLayer.endPoint = CGPointMake(1.0, 1.0)

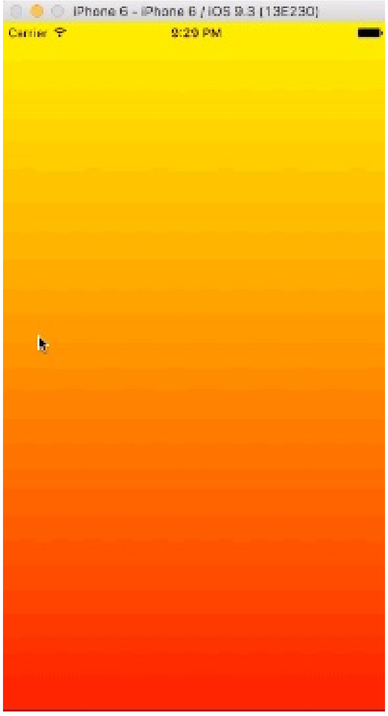
        case PanDirections.TopRightToBottomLeft.rawValue:
            gradientLayer.startPoint = CGPointMake(1.0, 0.0)
            gradientLayer.endPoint = CGPointMake(0.0, 1.0)

        case PanDirections.BottomLeftToTopRight.rawValue:
            gradientLayer.startPoint = CGPointMake(0.0, 1.0)
            gradientLayer.endPoint = CGPointMake(1.0, 0.0)

        default:
            gradientLayer.startPoint = CGPointMake(1.0, 1.0)
            gradientLayer.endPoint = CGPointMake(0.0, 0.0)
        }
    }
}
```

若panDirection为nil，则没有任何响应产生。

运行app，移动你的手指，渐变效果会随着你移动的方向而变化。



GitHub上的代码

How to Generate PDF using HTML Templates and UIPrintPageRenderer in iOS

如何使用HTML模板与UIPrintPageRenderer生成PDF

> - 原文链接：How to Generate PDF using HTML Templates and UIPrintPageRenderer in iOS
> - 原文作者：GABRIEL THEODOROPOULOS
> - 译者：yrq110

你曾被要求过生成包含app内容的PDF文档吗？

你想过怎么去做吗，或者说之前从没做过？

好吧，从提问开始可能有点非主流，不过上述的问题正是在这篇教程中要讨论的。生成一个iOS app的PDF文档的想法听起来挺无可救药的，实际上并不是。作为一个机智的开发者，总会找到最适合自己的方法并且是以最小的开支去实现目标。我不得不承认手工绘制一个PDF文档挺痛苦的，而且最后可能成为一项非生产性的任务。计算点，添加边，设置颜色，偏移等等，也许会很有趣，不过如果需要绘制的文档很复杂的话事情就会变得很麻烦。

目标是通过一个完全不同的方法来创建PDF文档，比手工绘制更加方便。这个方法是基于HTML模板的，大致可以简化为如下步骤:

1. 创建生成PDF所需的表单与content模板
2. 使用HTML模板产生content(也可用web view显示出来)
3. 导出PDF

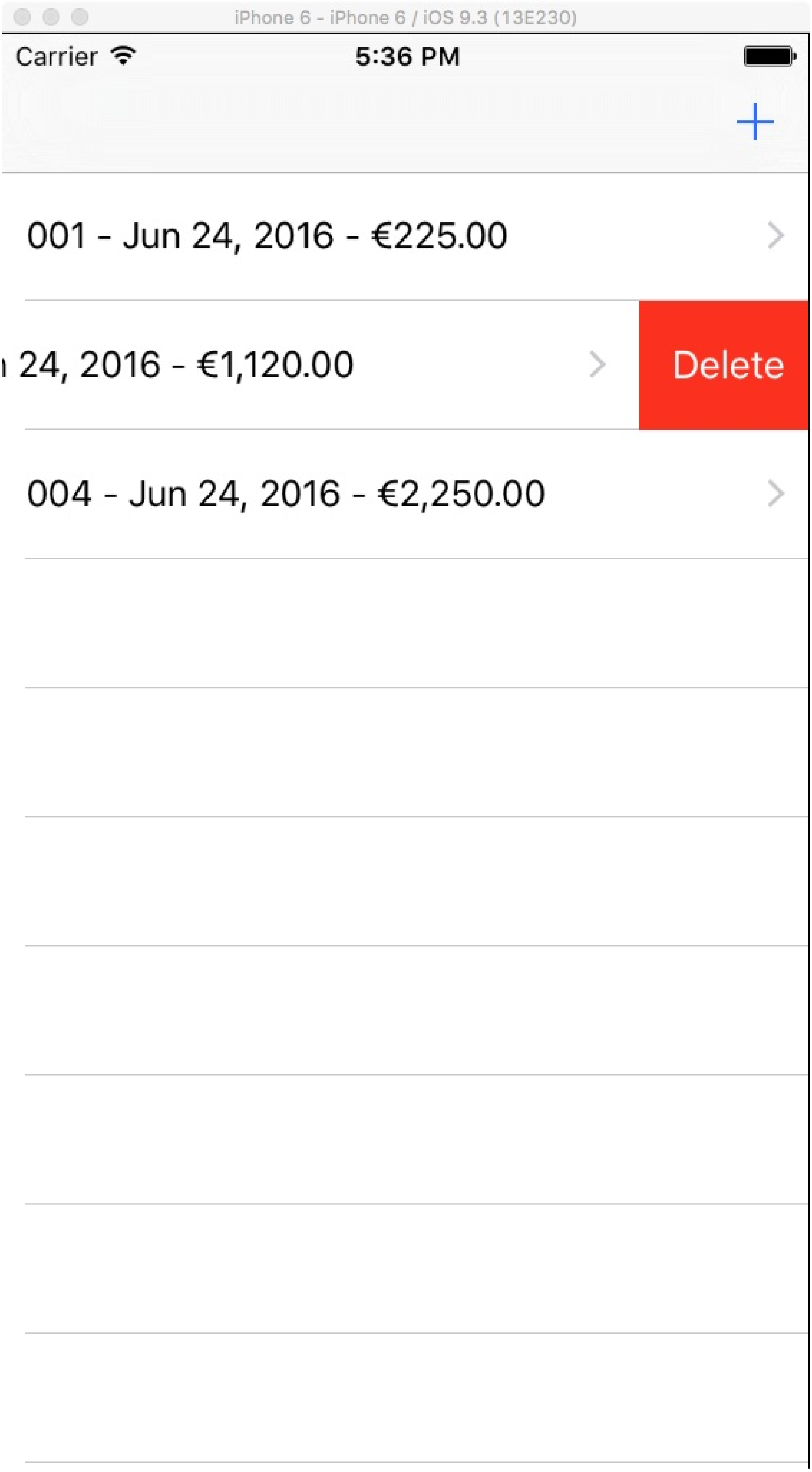
在最后一步，iOS会替你搞定所有麻烦事。

目录

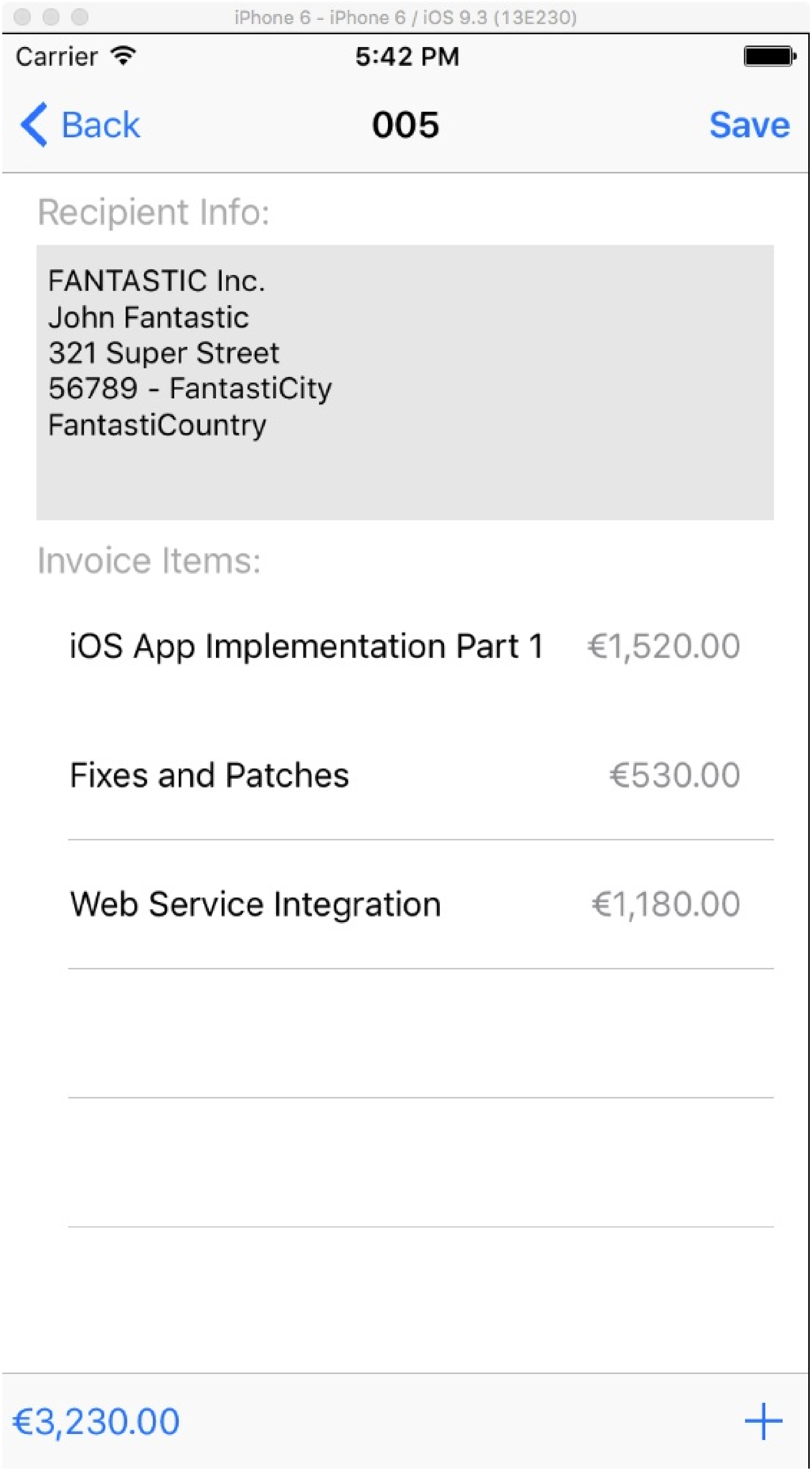
开始工程

使用一个发票生成器工具demo来进行教程中的演示。开始前请先下载开始工程然后用Xcode打开。

在工程中已经完成了一些必要的工作，InvoiceListViewController用来显示app中创建与保存的的发票列表，在这里通过右上角的加号按钮可以创建一个新的发票。点击任一列表项会进入发票信息的预览界面，可以导出成PDF格式的文档，注意，这部分功能还没有在开始工程中实现，需要我们在接下来完成它。还有，向左拖动列表项会出现删除选项，如下图:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
5:36 PM
001 - Jun 24, 2016 - €225.00
24, 2016 - €1,120.00
Delete
004 - Jun 24, 2016 - €2,250.00

前面说过，可以通过点击加号按钮创建新发票，会跳转到另一个视图控制器-CreatorViewController，如下图:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
5:42 PM
Back
005
Save
Recipient Info:
FANTASTIC Inc.
John Fantastic
321 Super Street
56789 - FantastiCity
FantastiCountry
Invoice Items:
iOS App Implementation Part 1
€1,520.00
Fixes and Patches
€530.00
Web Service Integration
€1,180.00
€3,230.00

导出之前，一个发票会生成所需要填写的一些变量值，有些是可以在视图控制器中手动设置的，有些事自动生成的，还有一些是硬编码，写死的。在demo app中需要手动设置的值有:

- 接收者信息，上面表格中的灰色区域。
- 发票条目, 每个条目都包含两部分: 所提供服务的描述与价格。为了方便起见，这里不考虑增值税。使用底部的加号按钮来添加条目。

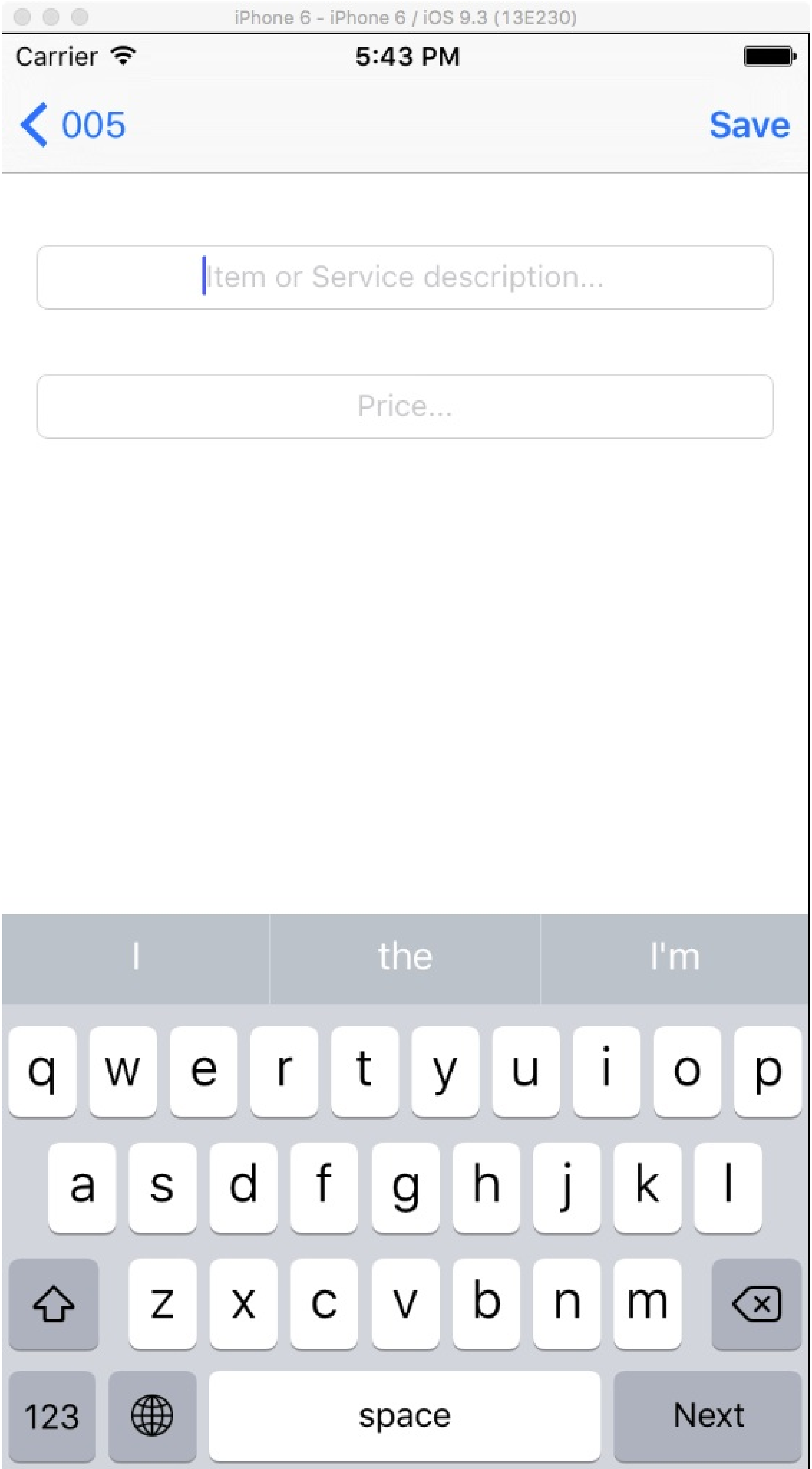
自动生成的值有:

- 发票编号(导航栏的标题数字)。
- 发票总价(底部工具栏的左侧显示).

最后是发票值中的硬编码部分:

- 发送者信息。
- 发票开具日期(设为空了如果你需要的话自行设置)。
- 支付方法。
- 发票的logo。

关于发票的条目，工程中有一个叫AddItemViewController的视图控制器可以用来输入简单的数据，有两个文本输入框和一个保存按钮，点击保存按钮就会添加条目到发票中，并返回到前一视图:


iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
5:43 PM
005
Save
Item or Service description...
Price...
I
the
I'm
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
123
space
Next

所有发票条目都添加进了一个字典数组中，每个字典包含两个值：描述与价格。将这个数组作为CreatorViewController中tableview的数据源，列出所有的条目。每当保存一个发票时，手动添加与自动生成的数据都会添加到字典中，并将一些数据返回到了InvoiceListViewController中，返回了如下这些值:

- 发票编号(字符串)
- 接收者信息(字符串)
- 总价(字符串)
- 发票条目(字典数组)

对于现有的代码乏善可陈，你需要做的就是看懂每个视图控制器中的流程与实现的细节。我想提的一点就是AppDelegate.swift文件，这里有三个方法：一个用于快速访问应用的delegate，一个用于得到文件路径，一个用于将总价用包含货币符号的字符串表示。这些方法已经在开始工程中用到了，之后还会使用它们。在AppDelegate中有一个currencyCode属性，默认设置的是“eur”(欧元)，可以根据自行情况设置。

点击InvoiceListViewController中列表中的一个发票，这个发票数据会传给PreviewViewController，在这里有一个预览发票的webview，预览被渲染成HTML文档的效果，有一个导出成PDF的按钮。这些函数在开始工程中没有，需要之后来实现它们，数据都是现成的可以直接使用。

HTML模板文件

接着我们要使用HTML模板来初始化一个发票，然后最终的HTML代码会导出成PDF文件。这里的主要逻辑就是先用占位符去填充HTML文件中的一些位置，然后再用真实的数据去替换，因此需要产生一个自定义的HTML表单来生成最终的结果。鉴于这篇教程的目的我们不会去创建任何发票的HTML模板，使用一个现成的模板(特别感谢作者)，并稍微修改了一下。

在开始工程中有三个HTML文件:

- invoice.html
- last_item.html
- single_item.html

第一个文件中包含产生除了条目外的整个发票代码。还有两个与条目有关的模板: single_item.html用来显示除了最后一个条目外的条目，last_item.html仅用来表示最后一个条目，这是因为最后一行的下面是有边框线的。

HTML模板文件中的占位符是被#号包含的特殊关键字。比如下面这段中有发票编号、签发日期与支付日期的占位符:

```
<td> Invoice #: #INVOICE_NUMBER<br>
#INVOICE_DATE#<br>
#DUE_DATE# </td>
```

> 注意: 虽然支付日期(due date)用了占位符，不过我们并不使用它，只用一个空字符串替换它，若你需要的话请自行更改。

可以在这三个HTML文件中发现很多占位符:

- #LOGO_IMAGE#
- #INVOICE_NUMBER#
- #INVOICE_DATE#
- #DUE_DATE#
- #SENDER_INFO#
- #RECIPIENT_INFO#
- #PAYMENT_METHOD#
- #ITEMS#
- #TOTAL_AMOUNT#
- #ITEM_DESC#
- #PRICE#

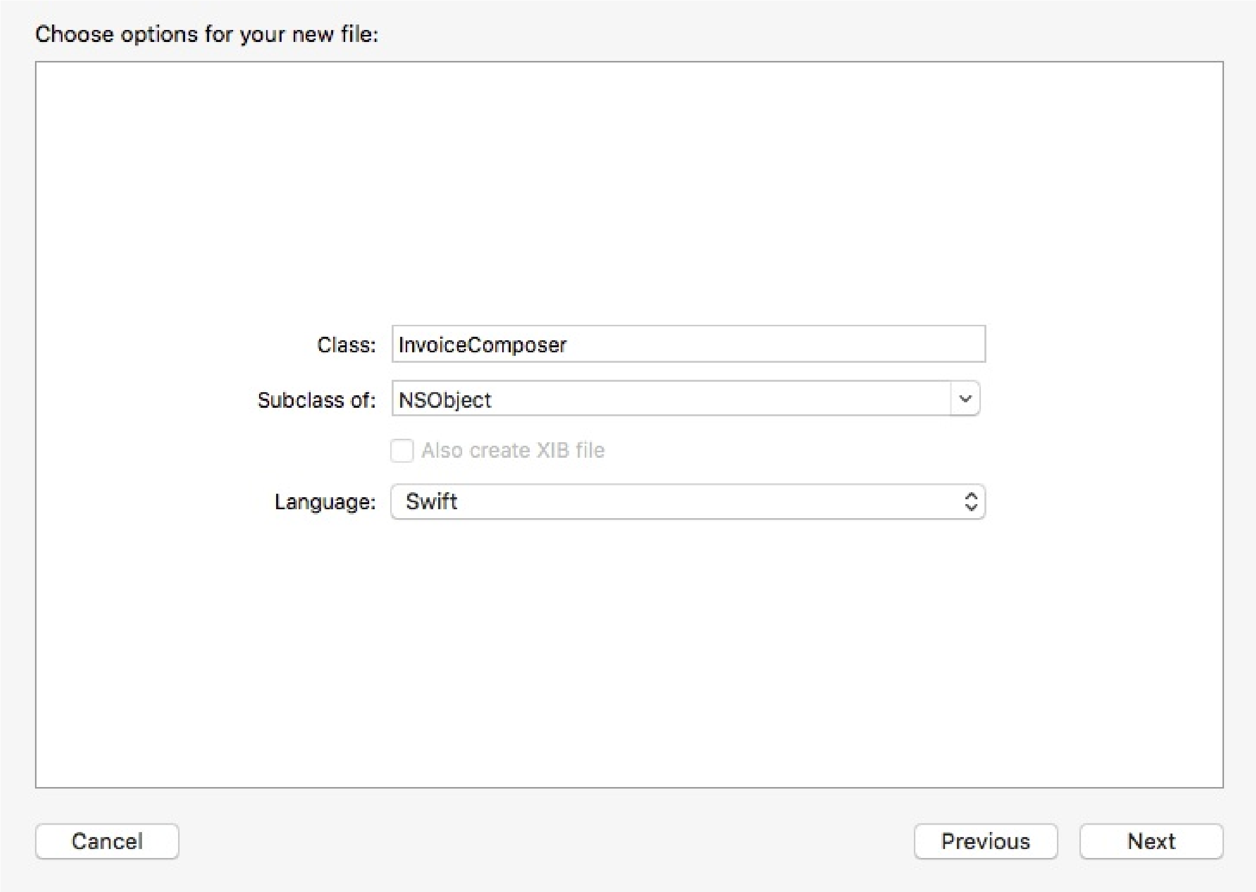
最后两个占位符只在single_item.html和last_item.html文件中有，#ITEMS#占位符被替换后每个item会使用其他2各HTML模板文件来创建各自的代码。

如你所见，准备多个HTML模板是为了创建自定义导出的表格，这并不难。进行完整个过程后你会发现，基于这些模板生成实际内容并提取成PDF文件是很简便并高效的。

组合内容

在熟悉了demo app与发票模板后，是时候来开始实现剩下所有关键点了。一开始，在第一个视图控制器(InvoiceListViewController)中使用HTML模板来生成一个发票的实际HTML内容，搞定这个后在PreviewViewController已存在的web view中加载生成的HTML代码并显示出来，验证一下结果。

这一部分最主要与关键的任务就是在发票的HTML模板中使用真实值替换掉占位符。这些数据是对应在InvoiceListViewController中所选择的发票数据，传递给PreviewViewController。后面你会看到替换占位符是个很简单的任务，在那之前让我们来创建一个新的类，用于生成真实的HTML内容与PDF格式的输出。创建一个新的Cocoa Touch类，使其继承自NSObject类，命名为InvoiceComposer，一路Next完成创建。



打开已存在的InvoiceComposer.swift文件，从声明一些属性开始 (包含常量与变量):

```
class InvoiceComposer: NSObject {

    let pathToInvoiceHTMLTemplate = NSBundle.mainBundle().pathForResource("invoice", ofType: "html")

    let pathToSingleItemHTMLTemplate = NSBundle.mainBundle().pathForResource("single_item", ofType: "html")

    let pathToLastItemHTMLTemplate = NSBundle.mainBundle().pathForResource("last_item", ofType: "html")

    let senderInfo = "Gabriel Theodoropoulos<br>123 Somewhere Str.<br>10000 - MyCity<br>MyCountry"

    let dueDate = ""

    let paymentMethod = "Wire Transfer"

    let logoImageURL = "http://www.appcoda.com/wp-content/uploads/2015/12/blog-logo-dark-400.png"

    var invoiceNumber: String!

    var pdfFilename: String!

}
```

前三个属性(pathToInvoiceHTMLTemplate, pathToSingleItemHTMLTemplate, pathToLastItemHTMLTemplate)指定了HTML模板文件的路径，当我们需要打开模板代码并修改时使用这个属性就会很方便。

如我之前所说，这个demo没有提供设置全部发票参数的选项(senderInfo, dueDate, paymentMethod, logoImageURL)，这些会被设置为硬编码，不过在一个真正的app中这些值应该能被用户设置与修改。最后一个要说的是我已经设置好了发票logo所使用的图片链接。你可以随便修改上述的属性(比如在senderInfo中填入你自己的信息)。

最后的invoiceNumber属性会保存发票的编号，并且在编辑发票的过程中一直是可见的。pdfFilename属性包含PDF文件的路径，这是之后需要的，现在不用管，只需声明即可。

除了上述这些属性，还需要添加默认的init()方法:

```
class InvoiceComposer: NSObject {

    ...

    override init() {
        super.init()
    }
}
```

现在来创建一个处理替换HTML模板文件中占位符的新方法，命名为renderInvoice，这里是需要的参数:

```
func renderInvoice(invoiceNumber: String, invoiceDate: String, recipientInfo: String,
items: [[String: String]], totalAmount: String) -> String! {

}
```

这些参数是在demo中创建一个发票时需要手动设置的，也是生成发票时所需要的数据。返回的字符串值是包含实际内容的最终的HTML代码。

来实现这个方法并进行第一项重要的任务。在下面的代码段中有两个关键的地方:首先是将invoice.html文件中的模板内容加载在字符串变量中以便修改。另一个是替换掉除了发票条目外的占位符，注释会帮助你理解:

```
func renderInvoice(invoiceNumber: String, invoiceDate: String, recipientInfo: String,
items: [[String: String]], totalAmount: String) -> String! {
    // Store the invoice number for future use.
    self.invoiceNumber = invoiceNumber

    do {
        // Load the invoice HTML template code into a String variable.
        var HTMLContent = try String(contentsOfFile: pathToInvoiceHTMLTemplate!)

        // Replace all the placeholders with real values except for the items.
        // The logo image.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#LOGO_IMAGE#",
 withString: logoImageURL)

        // Invoice number.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#INVOICE_NUMBE
R#", withString: invoiceNumber)

        // Invoice date.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#INVOICE_DATE#"
, withString: invoiceDate)

        // Due date (we leave it blank by default).
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#DUE_DATE#", w
ithString: dueDate)

        // Sender info.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#SENDER_INFO#"
, withString: senderInfo)

        // Recipient info.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#RECIPIENT_INF
O#", withString: recipientInfo.stringByReplacingOccurrencesOfString("\n", withString:
"<br>"))

        // Payment method.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#PAYMENT_METHO
D#", withString: paymentMethod)

        // Total amount.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#TOTAL_AMOUNT#"
, withString: totalAmount)

    }
    catch {
        print("Unable to open and use HTML template files.")
    }

    return nil
}
```

替换占位符的方法如上面那样简单，使用stringByReplacingOccurrencesOfString(...)字符串方法即可，将placeholder作为第一个参数，实际值作为第二个参数。替换大量的占位符可能有些乏味，不过毕竟这样做不难。

注意到所有工作都在一个do-catch语句中进行，从初始化文件路径的HTMLContent字符串开始，若出错了则返回nil，目前不会返回包含实际值的HTML内容，后面再说。

来关注发票条目的设置，由于它们的数量是可变的，因此我们使用一个循环来处理。对于除了最后一个的所有条目，打开single_item.html模板替换其中的占位符。对于最后一个条目，使用last_item.html模板，因为底部存在一条线。最终的HTML代码会添加到另一个字符串中(后面的allItems变量)。当这个字符串与所有条目的详细信息整合在一起后，会被用来替换HTMLContent字符串中的#ITEMS#占位符，在方法的最后会返回这个字符串。

在 `do` 的body中添加代码，如下:

```
func renderInvoice(invoiceNumber: String, invoiceDate: String, recipientInfo: String,
items: [[String: String]], totalAmount: String) -> String! {
     ...

    do {
        ...

        // The invoice items will be added by using a loop.
        var allItems = ""

        // For all the items except for the last one we'll use the "single_item.html"
template.
        // For the last one we'll use the "last_item.html" template.
        for i in 0..&lt;items.count {
            var itemHTMLContent: String!

            // Determine the proper template file.
            if i != items.count - 1 {
                itemHTMLContent = try String(contentsOfFile: pathToSingleItemHTMLTempl
ate!)
            }
            else {
                itemHTMLContent = try String(contentsOfFile: pathToLastItemHTMLTemplat
e!)
            }

            // Replace the description and price placeholders with the actual values.
            itemHTMLContent = itemHTMLContent.stringByReplacingOccurrencesOfString("#I
TEM_DESC#", withString: items[i]["item"]!)

            // Format each item's price as a currency value.
            let formattedPrice = AppDelegate.getAppDelegate().getStringValueFormattedA
sCurrency(items[i]["price"]!)
            itemHTMLContent = itemHTMLContent.stringByReplacingOccurrencesOfString("#P
RICE#", withString: formattedPrice)
```

```
            // Add the item's HTML code to the general items string.
            allItems += itemHTMLContent
        }

        // Set the items.
        HTMLContent = HTMLContent.stringByReplacingOccurrencesOfString("#ITEMS#", with
String: allItems)

        // The HTML code is ready.
        return HTMLContent
    }
    catch {
        print("Unable to open and use HTML template files.")
    }

    return nil
}
```

模板代码在适当的时候会被修改以产生一个包含实际内容的发票，下面我们要利用上述方法的结果做一些事情。

预览HTML

在创建发票的实际内容后，是时候来验证之前所做的工作是否达标了，因此，在这一节我们的目标是将HTML字符串加载到PreviewViewController中的web view中看看效果。注意这只是一个可选的步骤，在实际的app中没必要在导出PDF之前去使用一个web view去预览HTML，这里这么做是为了demo app的完整性。

打开PreviewViewController.swift文件，来到类顶部，首先声明一些新属性:

```
class PreviewViewController: UIViewController {

    ...

    var invoiceComposer: InvoiceComposer!

    var HTMLContent: String!

}
```

第一个对象的类是之前我们创建好的，这里进行一个简单的初始化即可。HTMLContent字符串变量在之后被用来存放时间的HTML代码。

接着按如下步骤创建一个新方法:

1. 初始化invoiceComposer对象。

2. 调用renderInvoice(...)方法生成发票的HTML代码。
3. web view加载HTML。
4. 将返回的HTML字符串分配给HTMLContent属性。

具体实现:

```
func createInvoiceAsHTML() {
    invoiceComposer = InvoiceComposer()
    if let invoiceHTML = invoiceComposer.renderInvoice(invoiceInfo["invoiceNumber"] as
! String,
                                                       invoiceDate: invoiceInfo["invoi
ceDate"] as! String,
                                                       recipientInfo: invoiceInfo["rec
ipientInfo"] as! String,
                                                       items: invoiceInfo["items"] as!
 [[String: String]],
                                                       totalAmount: invoiceInfo["total
Amount"] as! String) {

        webPreview.loadHTMLString(invoiceHTML, baseURL: NSURL(string: invoiceComposer.
pathToInvoiceHTMLTemplate!)!)
        HTMLContent = invoiceHTML
    }
}
```

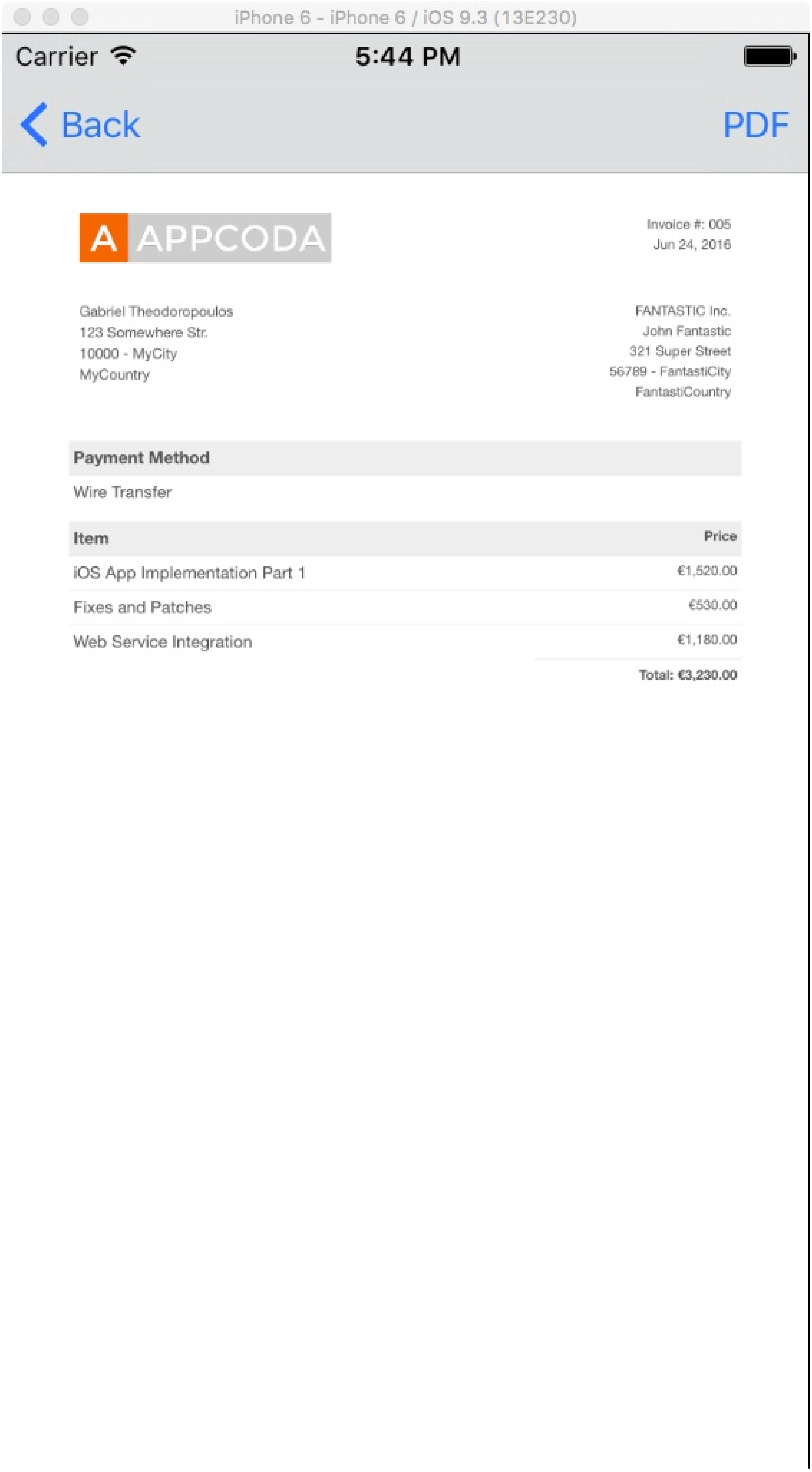
没啥难的地方，注意一下renderInvoice(...)方法的输入参数，当我们得到返回的实际HTML字符串后回家再加载web view中。

是时候调用新方法了:

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    createInvoiceAsHTML()
}
```

若想看看效果，运行app新建一个发票，在列表中选中点击，看看在web view中出现的发票，如下所示:



iPhone 6 - iPhone 6 / iOS 9.3 (13E230)
Carrier
5:44 PM
Back
PDF
APPCODA
Invoice #: 005
Jun 24, 2016
Gabriel Theodoropoulos
123 Somewhere Str.
10000 - MyCity
MyCountry
FANTASTIC Inc.
John Fantastic
321 Super Street
56789 - FantastiCity
FantastiCountry
Payment Method
Wire Transfer
Item
Price
iOS App Implementation Part 1
€1,520.00
Fixes and Patches
€530.00
Web Service Integration
€1,180.00
Total: €3,230.00

准备导出

已经搞定一半的工作了，现在来着手将一个发票生成PDF的过程。为了达到目的需要使用一个特殊的类: UIPrintPageRenderer，如果你没听说过它不要紧，这个类是可用于渲染打印内容的一种可靠方式，官方文档在这里，浏览一下查看更多的信息。

UIPrintPageRenderer类提供了多种绘制方法，一般情况下我们不需要去重写它。那些绘制方法只能被UIPrintPageRenderer的子类中的方法所重写，这些额外的努力是值得的，可以使我们灵活的控制导出的实际页面的内容: 比如页眉和页脚。在demo app中我们需要创建并使用它的子类，因为要绘制自定义的页眉和页脚(同时)。

回到Xcode中创建一个新的类，需要注意两件事:

1. 设置为UIPrintPageRenderer的子类.
2. 命名为CustomPrintPageRenderer.

准备好了后(已经在工程导航栏中看到CustomPrintPageRenderer.swift文件)，为下面的步骤做一些简单的准备。首先，指定A4页面的宽度和高度(单位为像素)，记住我们是想将发票提取成PDF，并且PDF是可以被正常打印出来的，所以限制页面大小是很重要的。

```
class CustomPrintPageRenderer: UIPrintPageRenderer {

    let A4PageWidth: CGFloat = 595.2

    let A4PageHeight: CGFloat = 841.8

}
```

上述值描述了精确的普通A4纸的宽度与高度。

指定纸张框架与CustomPrintPageRenderer类所绘制的打印区域是很必要的。在init()方法中进行这一步，使用上面定义的两个属性:

```
override init() {
    super.init()

    // Specify the frame of the A4 page.
    let pageFrame = CGRect(x: 0.0, y: 0.0, width: A4PageWidth, height: A4PageHeight)

    // Set the page frame.
    self.setValue(NSValue(CGRect: pageFrame), forKey: "paperRect")

    // Set the horizontal and vertical insets (that's optional).
    self.setValue(NSValue(CGRect: pageFrame), forKey: "printableRect")
}
```

在上面的代码段中，将纸张框架与打印区域设为了相同的值。然而也许你想给打印区域外面增加一些留白(小于纸张边界)，这样会得到更好的打印效果。这时，可以使用下面这行替换掉上述方法的最后一行代码:

```
   self.setValue(NSValue(CGRect: CGRectInset(pageFrame, 10.0, 10.0)), forKey: "printab
leRect")
```

上述代码在水平和垂直坐标方向各增加了10个像素的偏移，注意，这一节中所做的设置即使没有创建UIPrintPageRenderer的子类也是有效果的，换句话说，不论何时都不要忘记设置打印对象的纸张与打印区域。

生成PDF

说“打印成PDF”的意思实际上是绘制一些内容到PDF的图形上下文中去，一旦这一步发生后，那么绘制的内容就会被发送到一个打印机或者被保存到一个文件中。我们仅考虑第二种情况，将HTML内容绘制到一个PDF上下文中，接着将得到的结果保存到NSData对象中，最后将这个对象保存到一个文件(后缀.pdf文件)中。这个过程变化很多，不过步骤很简单。

打开InvoiceComposer.swift文件，实现exportHTMLContentToPDF(...)这个仅接受一个参数(提取成PDF所需的HTML内容)的新方法。在实现这个方法之前有必要介绍另一个与打印相关的概念——print formatter(UIPrintFormatter类). 苹果文档中有如下描述:

> UIPrintFormatter is an abstract base class for print formatters: objects that lay out custom printable content that can cross page boundaries. Given a print formatter, the printing system can automate the printing of the type of content associated with the print formatter.

这意味着对于我们来说只需要将HTML内容作为打印格式(print formatter)添加到打印页面的渲染器中，iOS的打印系统就会负责页面实际的布局与打印的内容。建议你看看官方的这个页面，说的很详细。有一点需要说明的是虽然UIPrintFormatter是一个抽象类，不过iOS SDK提

供了可以利用的具体的子类，比如说UIMarkupTextPrintFormatter，我们需要用这个类来添加HTML内容到打印页面的渲染器对象中去，更多细节与其它的子类可以在上面的官方链接中找到。

说完了，是时候实现新方法了:

```
func exportHTMLContentToPDF(HTMLContent: String) {
    let printPageRenderer = CustomPrintPageRenderer()

    let printFormatter = UIMarkupTextPrintFormatter(markupText: HTMLContent)
    printPageRenderer.addPrintFormatter(printFormatter, startingAtPageAtIndex: 0)

    let pdfData = drawPDFUsingPrintPageRenderer(printPageRenderer)

    pdfFilename = "\(AppDelegate.getAppDelegate().getDocDir())/Invoice\(invoiceNumber)
.pdf"
    pdfData.writeToFile(pdfFilename, atomically: true)

    print(pdfFilename)
}
```

逐步分析下:

- 首先初始化一个CustomPrintPageRenderer对象，使用它进行实际的绘制(也就是打印(printing))。
- 接着初始化一个UIMarkupTextPrintFormatter对象，输入HTML内容作为初始化的参数。
- 将page formatter添加到打印页面的渲染器对象中，addPrintFormatter(...)方法的第二个参数用来指定打印格式执行时的起始页面，这里我们设为0，并且只有1个页面。
- 下面进行实际的PDF绘制，drawPDFUsingPrintPageRenderer(...)是个自定义方法，后面会定义它。绘制PDF的结果会保存到pdfData对象中，pdfData实际上是个NSData对象。
- 将PDF数据保存到一个文件中。首先指定文件路径，同时给予一个合适的文件名(这里使用的是invoiceNumber属性)，然后将PDF数据写入该文件。
- 最后一步可选，打印出文件路径对在Finder中找到新创建的PDF文件是很有帮助的。运行app时，文件路径就会在控制器中打印出来，可以打开PDF进行预览并验证结果。

在一个更加复杂的app中可以使用多种打印格式对象，可以为每个格式指定不同的起始页面，不过现在在demo中不需要那么做，清晰地说明问题即可。

现在来实现自定义方法，执行绘制一个实际PDF上下文的工作。如你所见，下面用到了Core Graphics技术去实现，整个过程不长而且能望码生义:

```
func drawPDFUsingPrintPageRenderer(printPageRenderer: UIPrintPageRenderer) -&gt; NSData
! {
    let data = NSMutableData()

    UIGraphicsBeginPDFContextToData(data, CGRectZero, nil)

    UIGraphicsBeginPDFPage()

    printPageRenderer.drawPageAtIndex(0, inRect: UIGraphicsGetPDFContextBounds())

    UIGraphicsEndPDFContext()

    return data
}
```

首先初始化一个可变的数据对象，将导出的PDF数据写入其中。第二行命令代码去创建PDF的图形上下文。接着创建一个新的PDF页面，实际的绘制操作发生在下面这行:

```
printPageRenderer.drawPageAtIndex(0, inRect: UIGraphicsGetPDFContextBounds())
```

使用这一行，打印页面的渲染器对象就会在PDF上下文的框架中绘制其内容。注意在drawPageAtIndex(...)调用打印页面渲染器对象的绘制方法时，自定义的页眉与页脚也会被自动绘制出来。

最后关闭PDF图形上下文，返回绘制的最终数据到调用此方法的exportHTMLContentToPDF方法中去。

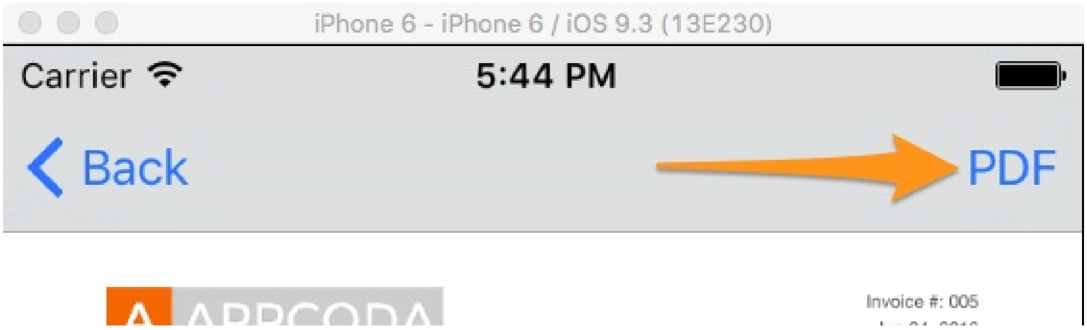
上述方法打印了一个单页面，若需要打印多个页面，则可以将PDF页面的创建与渲染放入一个循环中，若你想扩展这个demo app或想将多个页面打印到一个PDF文件中的话留心一下这一点。

这时与PDF导出有关的所有工作就告一段落了，下一节将要给你展示如果在打印页面中添加自定义的页眉和页脚，在那之前，先操作一下，看看导出PDF的效果。

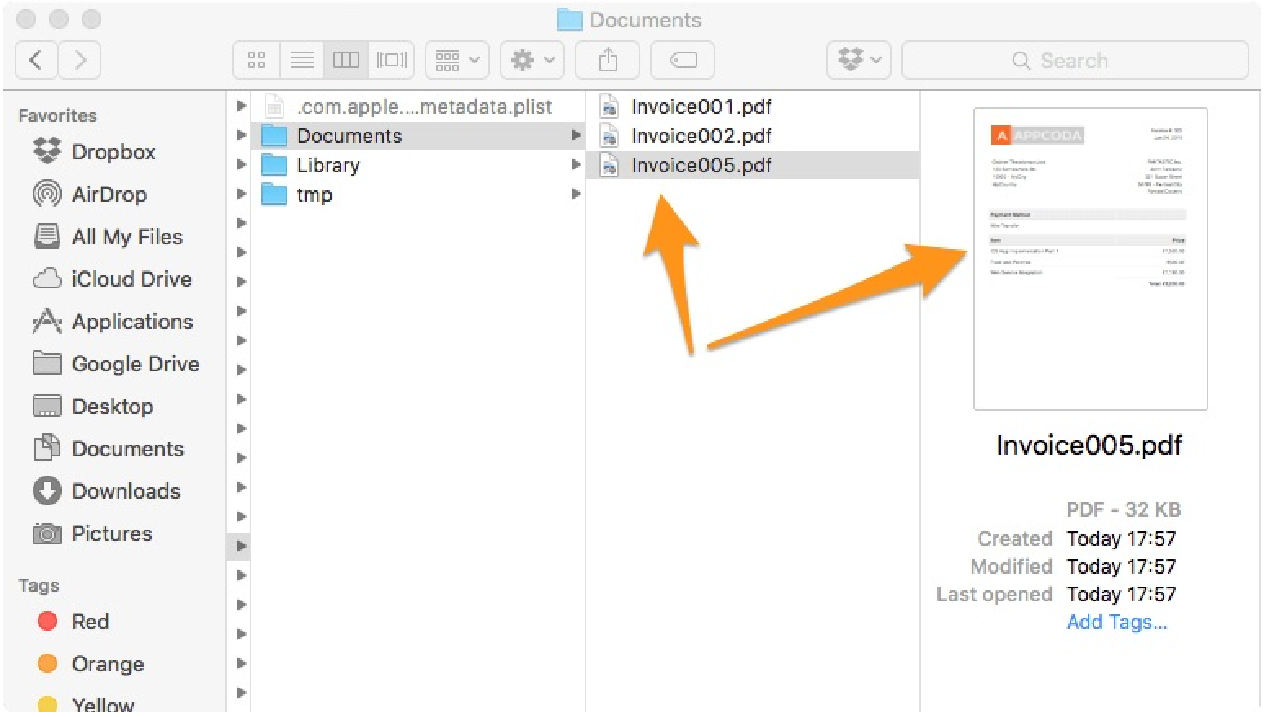
打开PreviewViewController.swift文件，定位到exportToPDF(...) IBAction方法中，添加如下代码，这样的话点击工具栏上的PDF按钮时就可以预览将发票提取成PDF后的效果了:

```
@IBAction func exportToPDF(sender: AnyObject) {
    invoiceComposer.exportHTMLContentToPDF(HTMLContent)
}
```

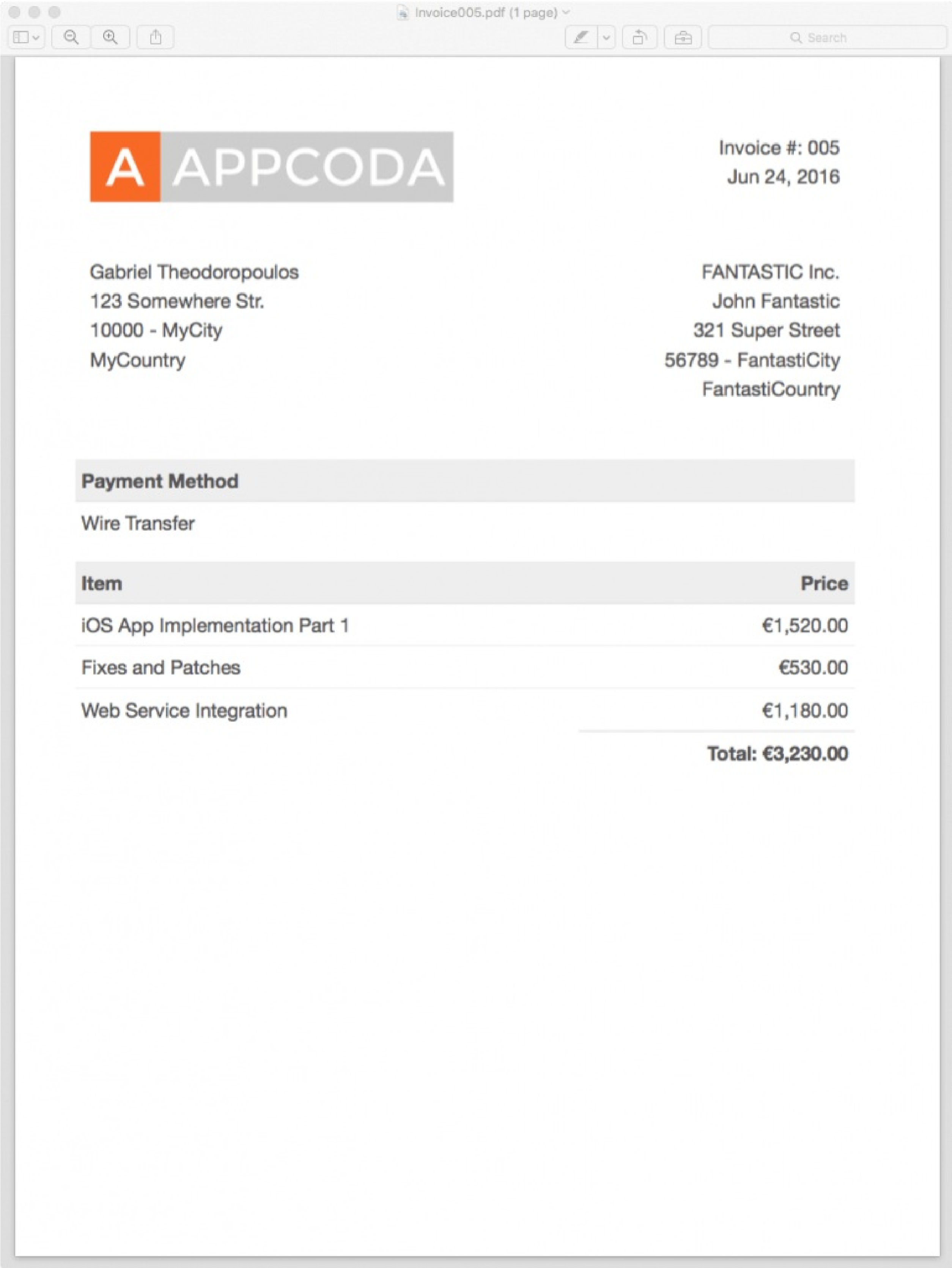
现在可以测试一个app，为了快捷我建议使用模拟器。选择一个发票来预览，点击右上角PDF按钮:



如此而为就会打印出PDF，当所有工作完成时，导出文件的路径会打印在控制器中。复制这个路径(不包括文件名)，打开一个新的Finder窗口，使用Shift-Command-G组合键，粘贴路径，则会看到刚刚生成的PDF文件，文件名为发票的编号。



双击它在Preview app中预览(或选择你喜欢的app):



APPCODA

Invoice #: 005
Jun 24, 2016

Gabriel Theodoropoulos
123 Somewhere Str.
10000 - MyCity
MyCountry

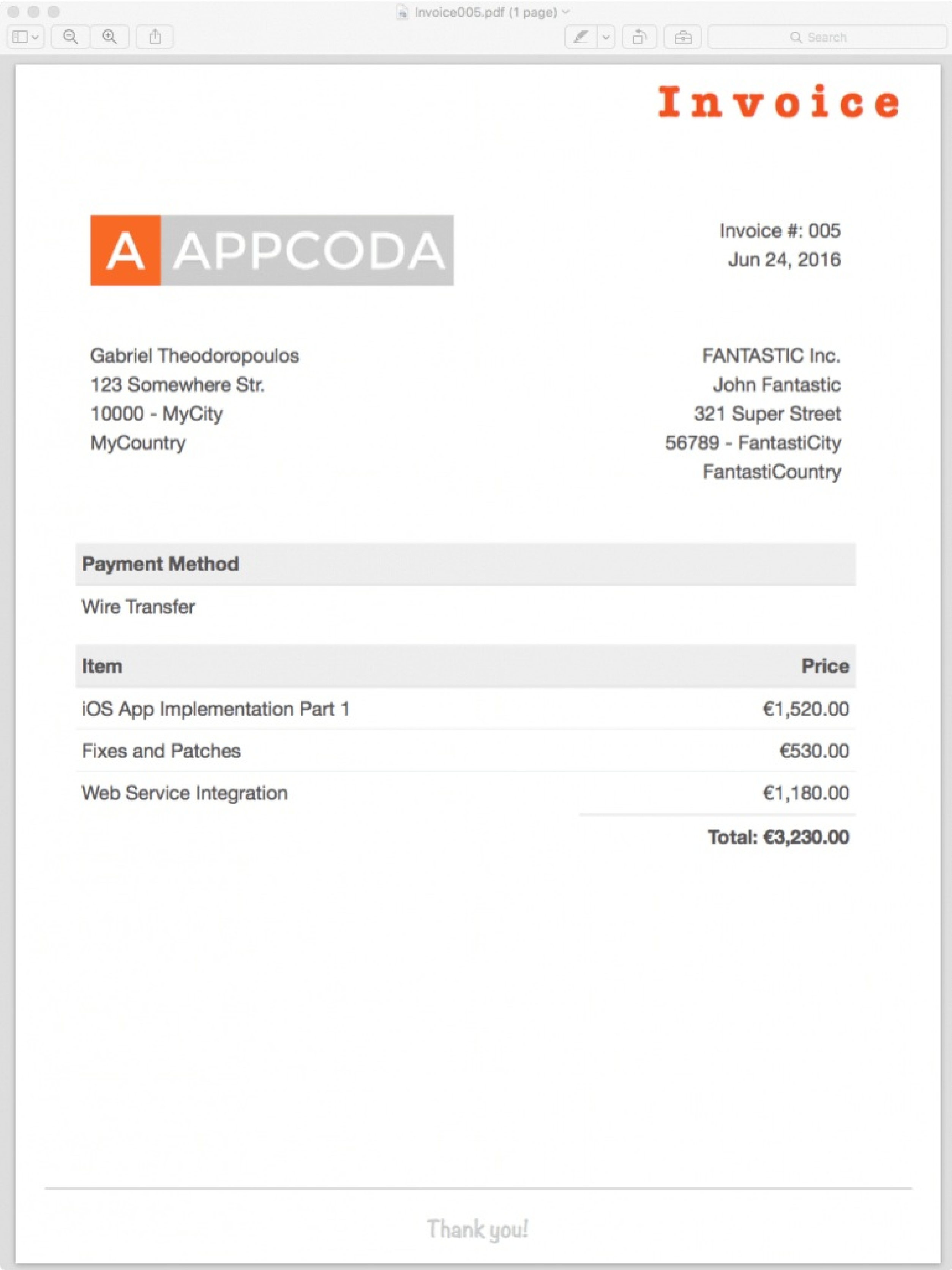
FANTASTIC Inc.
John Fantastic
321 Super Street
56789 - FantastiCity
FantastiCountry

Payment Method
Wire Transfer

Item	Price
iOS App Implementation Part 1	€1,520.00
Fixes and Patches	€530.00
Web Service Integration	€1,180.00
	Total: €3,230.00

绘制自定义页眉和页脚

现在来稍微扩展一下这个示例，给打印页面添加自定义的页眉和页脚，毕竟这是为何我们最初要创建UIPrintPageRenderer的子类。由于是自定义的，那么这意味着并不是HTML模板的一部分，不与HTML内容一同被渲染。我们想要实现的是在页面的右上角添加“Invoice”字样(作为页眉)，在页面底部添加短句“Thank you!”(页脚)与上方的水平线，目标如下图:



在实现页眉与页脚的细节之前，必须指定期望的页眉与页脚的高度。打开CustomPrintPageRenderer.swift文件，在init()方法中添加如下两行(注意两个属性都来自UIPrintPageRenderer类):

```
override init() {
    ...

    self.headerHeight = 50.0
    self.footerHeight = 50.0
}
```

从页面的页眉开始入手。重写UIPrintPageRenderer类的方法:

```
override func drawHeaderForPageAtIndex(pageIndex: Int, inRect headerRect: CGRect) {

}
```

将在函数体内做的步骤如下所述:

1. 指定在页眉中绘制的文本(单词“Invoice”)。
2. 指定文本格式的属性，例如字体、颜色与字母间距离。
3. 计算包含执行格式化后文本的矩形尺寸，指定距页面右侧边界的偏移。
4. 决定文本的绘制起点。
5. 绘制文本(最终)。

来实现一下上述的步骤，有一些注释可以帮助理解:

```
override func drawHeaderForPageAtIndex(pageIndex: Int, inRect headerRect: CGRect) {
    // Specify the header text.
    let headerText: NSString = "Invoice"

    // Set the desired font.
    let font = UIFont(name: "AmericanTypewriter-Bold", size: 30.0)

    // Specify some text attributes we want to apply to the header text.
    let textAttributes = [NSFontAttributeName: font!, NSForegroundColorAttributeName:
UIColor(red: 243.0/255, green: 82.0/255.0, blue: 30.0/255.0, alpha: 1.0), NSKernAttrib
uteName: 7.5]

    // Calculate the text size.
    let textSize = getTextSize(headerText as String, font: nil, textAttributes: textAt
tributes)

    // Determine the offset to the right side.
    let offsetX: CGFloat = 20.0

    // Specify the point that the text drawing should start from.
    let pointX = headerRect.size.width - textSize.width - offsetX
    let pointY = headerRect.size.height/2 - textSize.height/2

    // Draw the header text.
    headerText.drawAtPoint(CGPointMake(pointX, pointY), withAttributes: textAttributes
)
}
```

上述代码中有一点我没提到，就是getTextSize(...)方法的使用。如你所想，这个自定义方法用来计算并返回文本框架的尺寸。之所以把它写成一个额外的方法是因为在接下来绘制页脚时也需要用到它。

这是getTextSize(...)方法:

```
func getTextSize(text: String, font: UIFont!, textAttributes: [String: AnyObject]! = n
il) -&gt; CGSize {
    let testLabel = UILabel(frame: CGRectMake(0.0, 0.0, self.paperRect.size.width, foo
terHeight))
    if let attributes = textAttributes {
        testLabel.attributedText = NSAttributedString(string: text, attributes: attrib
utes)
    }
    else {
        testLabel.text = text
        testLabel.font = font!
    }

    testLabel.sizeToFit()

    return testLabel.frame.size
}
```

上面这个方法是个计算文本框架尺寸的通用方法。通过使用一个临时Label，赋予一个简单文本与字体或带属性的文本，然后使用sizeToFit()方法来计算其准确的尺寸。

来着手进行页面页脚的绘制，大致步骤跟页眉差不多，无需多言了。注意一下在文本的上方有一条水平线，并且文本颜色不同，文本的字符间距也不同:

```
override func drawFooterForPageAtIndex(pageIndex: Int, inRect footerRect: CGRect) {
    let footerText: NSString = "Thank you!"

    let font = UIFont(name: "Noteworthy-Bold", size: 14.0)
    let textSize = getTextSize(footerText as String, font: font!)

    let centerX = footerRect.size.width/2 - textSize.width/2
    let centerY = footerRect.origin.y + self.footerHeight/2 - textSize.height/2
    let attributes = [NSFontAttributeName: font!, NSForegroundColorAttributeName: UICo
lor(red: 205.0/255.0, green: 205.0/255.0, blue: 205.0/255, alpha: 1.0)]

    footerText.drawAtPoint(CGPointMake(centerX, centerY), withAttributes: attributes)
}
```

这段代码创建了“Thank you!”，不过没有添加水平线，需要使用更多的代码来画线:

```
override func drawFooterForPageAtIndex(pageIndex: Int, inRect footerRect: CGRect) {
    ...

    // Draw a horizontal line.
    let lineOffsetX: CGFloat = 20.0
    let context = UIGraphicsGetCurrentContext()
    CGContextSetRGBStrokeColor(context, 205.0/255.0, 205.0/255.0, 205.0/255, 1.0)
    CGContextMoveToPoint(context, lineOffsetX, footerRect.origin.y)
    CGContextAddLineToPoint(context, footerRect.size.width - lineOffsetX, footerRect.o
rigin.y)
    CGContextStrokePath(context)
}
```

现在就有一条水平线了！

在这一节结束前，关于页眉和页脚还有一点我想说一下，若你注意到的话会发现两个文本都是NSString对象，而不是String对象，这里有个特别的原因: drawAtPoint(...)方法是属于NSString类的绘制方法，如果使用String对象的话，需要先转换一下类型，如下所示:

```
(text as! NSString).drawAtPoint(...)
```

再次运行app，预览导出的PDF，这次就有页眉和页脚了。

可以在这里下载到完整的Xcode工程。

Building a Speech-to-Text App Using Speech Framework in iOS 10

使用iOS 10的Speech框架构建一个语音转文本app

> - 原文链接：Building a Speech-to-Text App Using Speech Framework in iOS 10
> - 原文作者：SAHAND EDRISIAN
> - 译者：yrq110

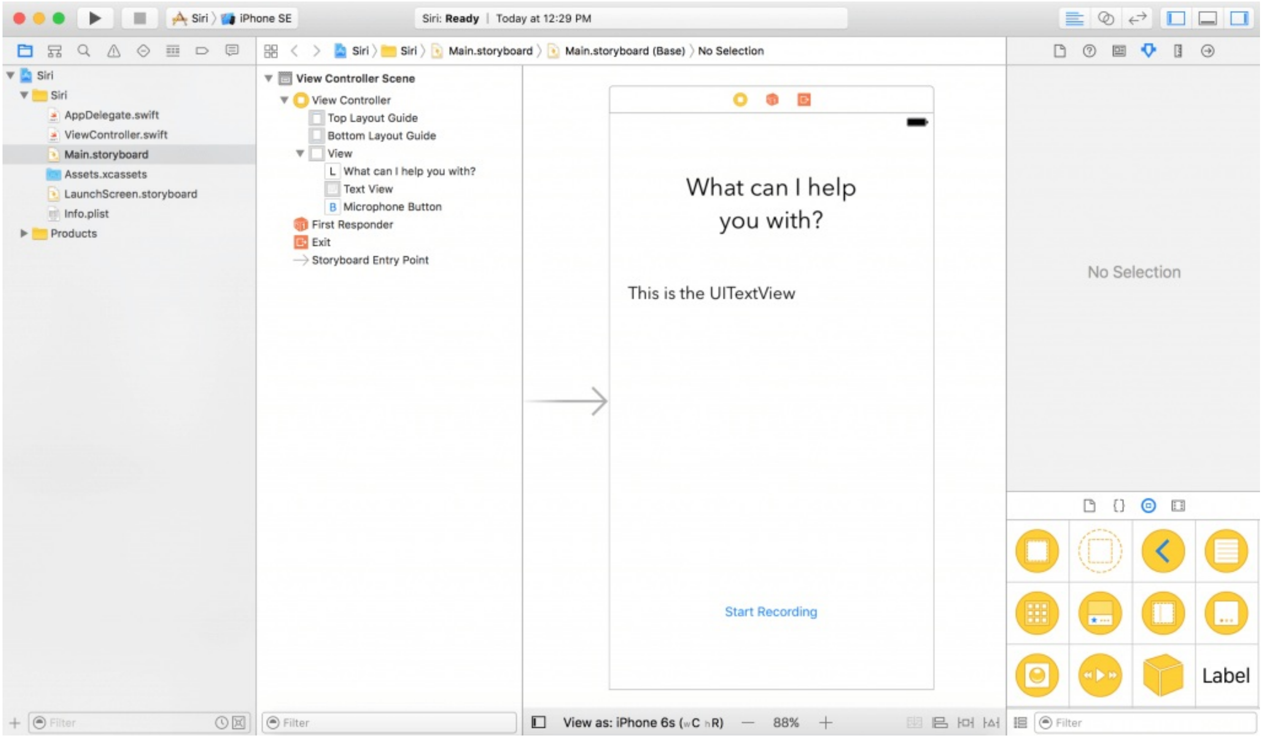
在WWDC 2016上苹果介绍了Speech框架-一个用于语音识别的API，其实Speech Kit就是Siri用于语音识别的框架。目前有少数可用的语音识别框架，不过那些要么太贵要么就是不太好。在这篇教程中，将会给你展示如何使用Speech Kit构建一个像Siri那样的语音转文本app。

目录

设计App的界面

来从创建一个新的iOS单视图应用开始，命名为 `SpeechToTextDemo` 。接着，打开 `Main.storyboard` 文件添加一个 `UILabel` ，一个 `UITextView` 和一个 `UIButton` 。

你的storyboard应该像这样:



下面需要在 `ViewController.swift` 中给 `UITextView` 和 `UIButton` 定义outlet变量，在demo中，将 `UITextView` 命名为“textView”， `UIButton` 命名为 “microphoneButton”，新建一个空的action方法作为microphone按钮被按下时的响应方法:

```
@IBAction func microphoneTapped(_ sender: AnyObject) {

}
```

若你不喜欢从零开始，直接下载这个开始工程然后跟着下面的教程走吧。

使用Speech框架

要使用Speech框架首先需要import然后满足 `SFSpeechRecognizerDelegate` 协议。来先import框架然后将这个协议添加到 `ViewController` 中，这样做之后 `ViewController.swift` 应变成下面这样:

```
import UIKit
import Speech

class ViewController: UIViewController, SFSpeechRecognizerDelegate {


    @IBOutlet weak var textView: UITextView!
    @IBOutlet weak var microphoneButton: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()

    }

    @IBAction func microphoneTapped(_ sender: AnyObject) {

    }

}
```

使用授权

在使用speech框架进行语音识别之前，首选需要请求用户许可，这是因为这个识别的过程不是在iOS设备上进行的而是在苹果的服务器上。所有音频数据都会发送给苹果后端去处理，因此需要写一些代码来得到用户的授权。

在viewDidLoad方法中进行speech框架的授权，用户必须允许app使用输入音频与语音识别。首先声明一个 `speechRecognizer` 变量:

```
private let speechRecognizer = SFSpeechRecognizer(locale: Locale.init(identifier: "en-US"))  //1
```

如下修改viewDidLoad方法:

```
override func viewDidLoad() {
    super.viewDidLoad()

    microphoneButton.isEnabled = false  //2

    speechRecognizer.delegate = self  //3

    SFSpeechRecognizer.requestAuthorization { (authStatus) in  //4

        var isButtonEnabled = false

        switch authStatus {  //5
        case .authorized:
            isButtonEnabled = true

        case .denied:
            isButtonEnabled = false
            print("User denied access to speech recognition")

        case .restricted:
            isButtonEnabled = false
            print("Speech recognition restricted on this device")

        case .notDetermined:
            isButtonEnabled = false
            print("Speech recognition not yet authorized")
        }

        OperationQueue.main.addOperation() {
            self.microphoneButton.isEnabled = isButtonEnabled
        }
    }
}
```

1. 首先新建一个SFSpeechRecognizer常量，设置locale参数为en-US——指明用户将说的是何种语言。这个就是用来处理语音识别的对象。
2. 默认情况下禁用microphone按钮直到speechRecognizer被开启。
3. 接着设置speechRecognizer的委托为self，这里指ViewController。
4. 之后，必须请求语音识别的授权，通过调用SFSpeechRecognizer.requestAuthorization。
5. 最后检查授权的状态，如果被授权则启用microphone按钮，否则输出错误信息并禁用microphone按钮。

现在你可能会认为运行app时会弹出授权提醒的窗口，然而你错了。如果你运行了就会崩溃，这是为何?

提供授权信息

苹果要求所有授权都需要有一个来自app的自定义信息(授权描述)。在语音识别的情况下，我们必须得到两样事物的授权:

1. 麦克风
2. 语音识别

为了自定义这个信息，需要在 `info.plist` 文件中提供这些信息。

打开 `info.plist` 文件的源代码。首先在 `info.plist` 上右击，选择Open As > Source Code。最后复制下面的代码粘贴在 `</dict>` 标签前。

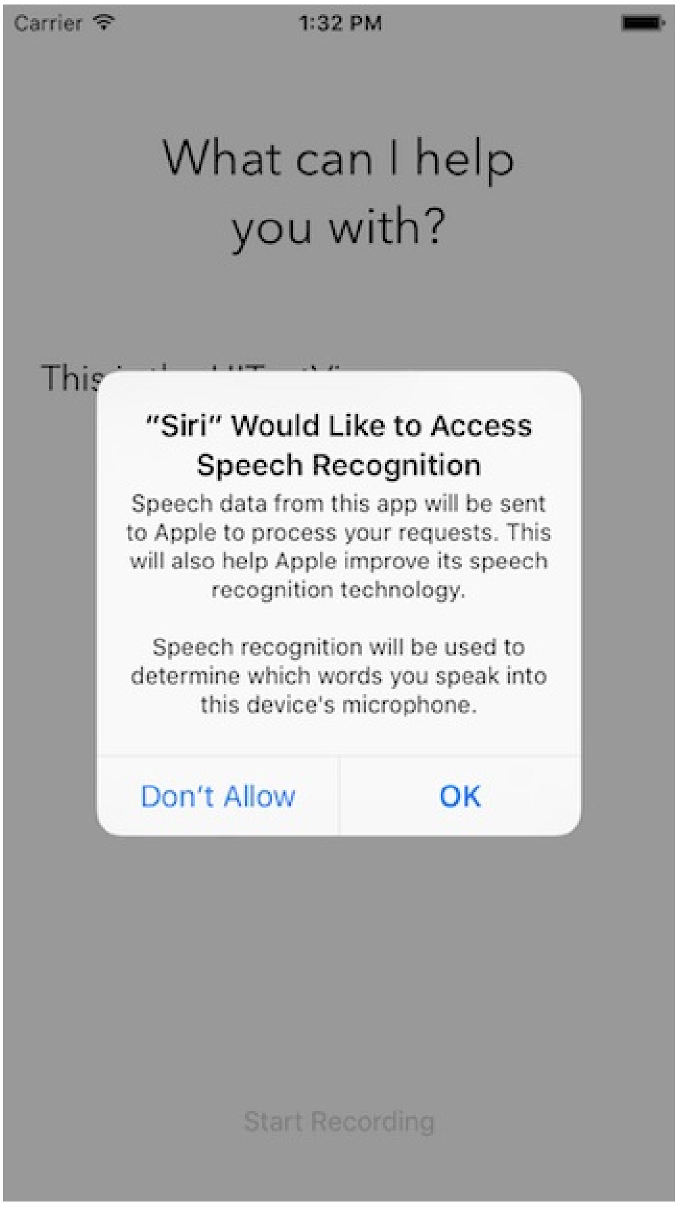
```
<key>NSMicrophoneUsageDescription</key>  <string>Your microphone will be used to record your speech when you press the \&quot;Start Recording\&quot; button.</string>

<key>NSSpeechRecognitionUsageDescription</key>  <string>Speech recognition will be used to determine which words you speak into this device\&apos;s microphone.</string>
```

在 `info.plist` 中添加了两个key:

1. `NSMicrophoneUsageDescription` – 授权音频输入的自定义信息。注意音频输入的授权只发生在用户点击microphone按钮时。
2. `NSSpeechRecognitionUsageDescription` – 语音识别的自定义信息。

添加完这些记录后点击Run按钮编译并运行app。



> 注意: 若当你运行时没有看到输入音频的授权，是因为app是在模拟器上运行的。iOS模拟器是不会访问Mac的麦克风的。

处理语音识别

现在我们已经完成了用户授权部分，来着手语音识别的实现，从在 `ViewController` 中定义如下对象开始:

```
private var recognitionRequest: SFSpeechAudioBufferRecognitionRequest?
private var recognitionTask: SFSpeechRecognitionTask?
private let audioEngine = AVAudioEngine()
```

1. 这个对象用来处理语音识别请求，给语音识别器提供一个音频输入。
2. 保存识别请求的结果，使用这个对象可以方便的取消或停止任务。

3. 这是你的 `audioEngine` ，负责提供输入音频。

新建一个 `startRecording()` 方法。

```
func startRecording() {

    if recognitionTask != nil {
        recognitionTask?.cancel()
        recognitionTask = nil
    }

    let audioSession = AVAudioSession.sharedInstance()
    do {
        try audioSession.setCategory(AVAudioSessionCategoryRecord)
        try audioSession.setMode(AVAudioSessionModeMeasurement)
        try audioSession.setActive(true, with: .notifyOthersOnDeactivation)
    } catch {
        print("audioSession properties weren't set because of an error.")
    }

    recognitionRequest = SFSpeechAudioBufferRecognitionRequest()

    guard let inputNode = audioEngine.inputNode else {
        fatalError("Audio engine has no input node")
    }

    guard let recognitionRequest = recognitionRequest else {
        fatalError("Unable to create an SFSpeechAudioBufferRecognitionRequest object")
    }

    recognitionRequest.shouldReportPartialResults = true

    recognitionTask = speechRecognizer.recognitionTask(with: recognitionRequest, resul
tHandler: { (result, error) in

        var isFinal = false

        if result != nil {

            self.textView.text = result?.bestTranscription.formattedString
            isFinal = (result?.isFinal)!
        }

        if error != nil || isFinal {
            self.audioEngine.stop()
            inputNode.removeTap(onBus: 0)

            self.recognitionRequest = nil
            self.recognitionTask = nil

            self.microphoneButton.isEnabled = true
        }
```

```
    })

    let recordingFormat = inputNode.outputFormat(forBus: 0)
    inputNode.installTap(onBus: 0, bufferSize: 1024, format: recordingFormat) { (buffe
r, when) in
        self.recognitionRequest?.append(buffer)
    }

    audioEngine.prepare()

    do {
        try audioEngine.start()
    } catch {
        print("audioEngine couldn't start because of an error.")
    }

    textView.text = "Say something, I'm listening!"

}
```

这个函数会在开始录音按钮被按下时调用，主要功能是开启语音识别与麦克风，来逐行分析下:

1. 3-6行 – 检查 `recognitionTask` 是否在运行中，如果是则取消。
2. 8-15行 – 创建一个 `AVAudioSession` 对象为录音做准备，将category设置为Record(录制音频。静音不停止)，mode设置为Measurement(减少设备在处理音频I/O 的信号量时的影响)，然后启用。注意设置的这些属性有可能抛出异常，因此必须把它们放在一个try catch语句中。
3. 17行 – 初始化 `recognitionRequest` ，新建 `SFSpeechAudioBufferRecognitionRequest` 对象，之后会使用它来将音频数据递送给苹果服务器。
4. 19-21行 – 检查 `audioEngine` 是否有录音的音频输入，没有则报告一个重要错误。
5. 23-25行 – 检查 `recognitionRequest` 对象是否被初始化是否为空。
6. 27行 – 告诉 `recognitionRequest` 分段输出用户说话的语音识别结果。
7. 29行 – 调用 `speechRecognizer` 的 `recognitionTask` 方法开始识别，这个函数有一个完成句柄，每次识别引擎收到输入时都会调用这个句柄，在被取消或停止时会返回一个最终的记录。
8. 31行 – 定义一个决定识别是否终止的布尔值。
9. 35行 – 若结果非空，则设置 `textView.text` 属性为结果中最好的记录。若结果为最终结果则设 `isFinal` 为true。
10. 39-47行 – 若没有任何错误或者有最终结果了，则停止 `audioEngine` 、 `recognitionRequest` 与 `recognitionRequest` ，同时启用开始录音按钮。
11. 50-53行 – 给 `recognitionRequest` 添加一个音频输入，注意在启动 `recognitionTask` 之后添加音频输入是可以的。Speech框架会在添加音频输入的同时开始识别。
12. 55行 – 准备并开启 `audioEngine` (音频输入)。

触发语音识别

我们需要确保当新建语音识别任务时它的功能是可用的，因此需要在 `ViewController` 中添加一个委托方法。当语音识别不可用或者状态发生改变时，需要设置 `microphoneButton.enable` 属性。实现SFSpeechRecognizerDelegate协议中的availabilityDidChange方法，如下所示:

```
func speechRecognizer(_ speechRecognizer: SFSpeechRecognizer, availabilityDidChange av
ailable: Bool) {
    if available {
        microphoneButton.isEnabled = true
    } else {
        microphoneButton.isEnabled = false
    }
}
```

当可用性改变时会调用这个方法。若语音识别可用，则启用录音按钮。

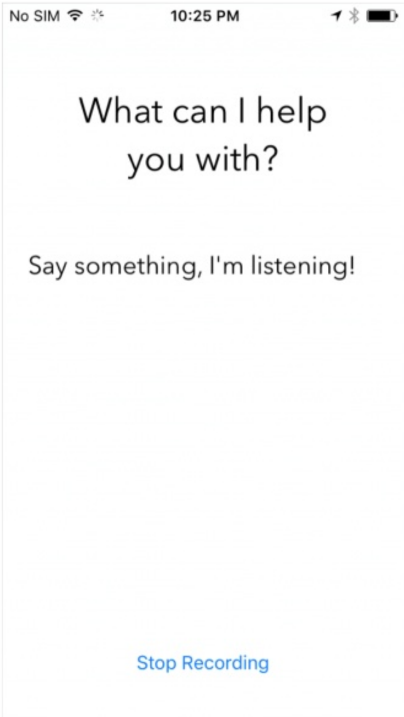
最后一件事就是更新响应事件的方法 `microphoneTapped(sender:)`：

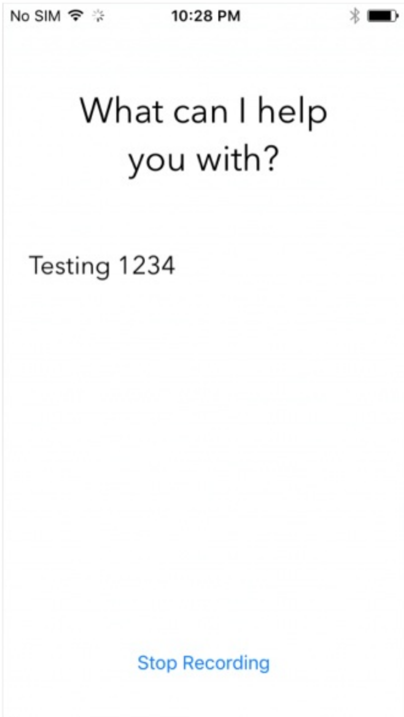
```
@IBAction func microphoneTapped(_ sender: AnyObject) {
    if audioEngine.isRunning {
        audioEngine.stop()
        recognitionRequest?.endAudio()
        microphoneButton.isEnabled = false
        microphoneButton.setTitle("Start Recording", for: .normal)
    } else {
        startRecording()
        microphoneButton.setTitle("Stop Recording", for: .normal)
    }
}
```

在这个函数中必须检查 `audioEngine` 是否正在运行，若运行中则应该停止 `audioEngine` ，终止 `recognitionRequest` 的音频输入，禁用 `microphoneButton` ，设置按钮标题为"开始录音"。

若 `audioEngine` 可用则app会调用 `startRecording()` 并设置按钮标题为"停止录音"。

棒！完成了准备工作，下面来测试app，将app部署到iOS 10设备上，点击开始录音按钮，说点什么吧！





注意:

1. 苹果限制每台设备的识别。不太清楚详情，你可以联系苹果获得更多信息。
2. 苹果限制每个app的识别。
3. 若你经常被限制，联系一下苹果，它们会解决的。
4. 语音识别很费电，很耗资源。
5. 语音识别每次仅持续1分钟左右。

总结

期待开发者们利用新的API作出更多惊艳的app~

可以看看WWDC 2016 session 509获取更多的信息。

在Github上查看完整的工程。

How To Create an Uber Splash Screen

如何创建一个Uber启动动画

How To Create an Uber Splash Screen

Building iOS Apps with Xamarin and Visual Studio

在Xamarin与Visual Studio中构建iOS应用

> - 原文链接 : Building iOS Apps with Xamarin and Visual Studio
> - 原文作者 : Bill Morefield
> - 译者 : yrq110

> PS：本篇教程使用的是C#，并非Swift。

在创建iOS app时，开发者通常会选择苹果提供的语言与集成开发环境: Objective-C / Swift与Xcode，然而这并不是唯一的选择，可以使用多种语言与框架进行iOS app的创建。

Xamarin是一个比较流行的选择，一个跨平台的框架，允许你使用C#和Visual Studio进行iOS, Android, OS X与Windows app的开发。Xamarin的一个主要优势在于可以在iOS与Android间共用一个app代码。

Xamarin跟其他跨平台框架比有一个很大的优点: 使用Xamarin的话你的工程会编译成本地代码(native code)，并可以使用内部的本地API，这意味着一个书写良好的Xamarin app与Xcode中的app是没有区别的。想了解更多细节可以看看这篇文章Xamarin vs. Native App Development 。

Xamarin在之前存在一个巨大的缺点: 价格。每个平台每年1000美元的高昂价格也许会使你放弃日常的拿铁与卡布奇诺去支付它...没有咖啡的编程是很危险的。因为这个高昂的价格，直到最近Xamarin都还是主要吸引一些有着庞大预算的企业项目的注意。

不过最近发生了变化，Microsoft收购了Xamarin并宣布它将支持全版本的Visual Studio，包括免费的Community Edition，方便了个人开发者与小型组织。

免费? 是个值得庆祝的价格!

更多的咖啡钱!

除了开销外, Xamarin的其它优点有:

- 利用现有的C#库与工具来构建移动app。
- 在不同平台间复用app代码。
- 在ASP.Net后端与面向用户的app间共享代码。

Xamarin同时提供了多种工具来选择，取决于你的需求。使用Xamarin Forms来最大化跨平台的代码复用，无需针对平台来指定功能与特殊的自定义接口，非常有效。

若你的app需要某些平台独有的特性或设计，使用Xamarin.iOS、Xamarin.Android与其他针对平台的模块来直接使用本地API与框架来进行交互。这些模块提供灵活创建自定义用户接口的功能，同时也允许跨平台共享代码。

在这篇教程中将会使用**Xamarin.iOS**来创建一个用来显示用户照片的iPhone app。

无需任何iOS或Xamarin开发经验，不过需要你掌握一些C#的基本知识使知识的收益最大化。

目录

入门

为了使用Xamarin和Visual Studio开发iOS应用，理论上需要两台机器:

1. 一台**Windows** 用来运行Visual Studio与写项目代码。
2. 一台安装了**Xcode**的**Mac** 作为构建主机(build host)。可以不是一个专门用来构建的机子，只要在Windows测试与开发时保持网络畅通就行。

如果这两台机子离的比较近就再好不过了，当你在Windows上构建运行时，Mac上的iOS模拟器就会加载对应的项目。

我知道你会说，“我没两台机子咋整?!”

- 对于仅使用**Mac**的用户 Xamarin为OS X提供了一个IDE，不过在这篇教程中只关注Visual Studio。因此若你想跟着教程走可以在Mac上运行一个Windows虚拟机，可以使用VMWare Fusion或免费、开源的VirtualBox来创建虚拟机。

若使用Windows虚拟机，需要确保Windows与Mac间保持正常的网络连接。通常会在Windows中ping Mac的IP地址测试一下。

- 对于仅使用**Windows**的用户 现在去买个Mac，我会等你！:]也可以使用像MacinCloud或Macminicolo这种提供远程访问服务的Mac主机来构建。

这篇教程中假定你分别使用Mac与Windows电脑，别担心，在Mac上使用Windows虚拟机也是可以的。

安装Xcode与Xamarin

若还没准备好Xcode的话先在你的Mac上下载并安装Xcode，跟安装其它App Store上的app一样，不过大小上G了，需要花一些时间。

吃点曲奇休息片刻!

Xcode安好后，在Mac上下载Xamarin Studio，需要提供你的邮箱地址来下载。

下载完成后打开安装包双击**Xamarin.app**安装，接受条款一路continue。

安装器会搜索已存在的工具与当前平台版本，并列出一个当前开发环境的详细列表。确保选择的是**Xamarin.iOS**，点击**Continue**。

接着是安装项目的确认列表，点击**Continue**执行，会出现一个Summary与一个启动Xamarin Studio的选项，点击**Quit**完成安装。

安装Visual Studio与Xamarin

这篇教程中可以使用任何版本的Visual Studio，包括免费的Community Edition。有些特性在Community Edition中没有，不过这并不影响你开发出复杂的app来。

查看Windows系统的Visual Studio最低硬件需求，为了有一个好的开发体验，至少要3G的内存。

若还未安装Visual Studio的话，在Community Edition网站点击绿色的**Download Community 2015**按钮下载**Community Edition**安装器。

运行安装器开始安装，选择自定义安装选项，在列表中展开**Cross Platform Mobile Development**，选择C#/.NET (Xamarin v4.0.3)(v4.0.3是教程在撰写时的版本，未来也许会变化)。

点击Next等待安装完成，会花点儿时间 :]

若你已准备好Visual Studio了不过还没有Xamarin工具，打开Windows系统的**Programs and Features**找到Visual Studio 2015，访问其安装程序，选择修改，会在Cross Platform Mobile Development下面看到**C#/.NET (Xamarin v4.0.3)**，选择它点击更新进行安装。

安了一大堆东西啊，现在搞定所有需要的东西了!

创建App

打开Visual Studio选择**File\New\Project**，在Visual C#下展开**iOS**，选择**iPhone**，右侧选择**Single View App**模板。这个模板会创建一个具有单视图控制器的app。

Name和**Solution Name**都输入**ImageLocation**，选择一个app的保存路径点击**OK**创建工程。

Visual Studio需要你准备好作为Xamarin构建主机的Mac:

1. 在Mac上打开**System Preferences**选择**Sharing**。
2. 开启**Remote Login**。
3. 选择**Allow access**下的**Only these users**，添加一个用户用来访问Mac上的Xcode和Xamarin。
4. 关闭设置回到Windows电脑。

回到Visual Studio中，会被询问是否将Mac作为构建主机。选择Mac点击**Connect**，输入用户名与密码点击**Login**。

可以通过工具栏确认连接状态。

从解决平台的下拉菜单中选择iPhone Simulator，这时在构建主机中会自动选择一个模拟器，可以通过点击当前模拟器旁边的小箭头来改变模拟器设备。

点击绿色的**Debug**箭头或者快捷键**F5**来构建并运行程序。

app会编译并执行，不过你在Windows上看不到它的运行情况，需要在Mac上看，这就是为何两台机子距离较近比较好 :]

在最近的Evolve会议上，Xamarin宣布iOS Simulator Remoting很快就能实现app运行在苹果的iOS 模拟器就好像是运行在Windows上的模拟器一样，不过现在你需要操作Mac上的模拟器。

能看到一个显示启动界面的模拟器，然后出现一个空的视图。恭喜！Xamarin设置成功。

点击红色停止按钮(快捷键Shift+F5)来停止app。

创建Collection View

app会在一个Collection View中显示用户照片的缩略图，Collection View是一个使用网格显示一些物件的iOS控件。

在**Solution Explorer**中打开**Main.storyboard**，编辑app的故事板，其中包含app的“场景”。

打开**Toolbox**在文本框中输入**collection**过滤出部件。将**Data Views**下面的**Collection View**对象拖拽到空视图上面。

选择collection view，应该会看到视图的每条边上都有空心圆，若看到的是**T**型图案，再次点击切换到空心圆。

点击并拖动每个圆到视图的边界直接出现蓝色线条，这时松开鼠标边界就会与这个位置对齐。

现在需要设置collection view的Auto Layout Constraints，这会告知app当设备旋转时如何调整视图的尺寸。在故事板上方的工具栏中，点击CONSTRAINTS标签旁的绿色按钮添加信号，这会给collection view自动添加constrains。

生成的constrains基本上没问题，因此你无需修改它们。在**Properties**窗口，切换到**Layout**子视图然后滚动到下面的**Constraints**区域。

由边界生成的两个constraints是正确的，不过高度和宽度constraints却有问题，点击每栏后面的**X**删除Width与Height constraints。

注意若collection view变成了橘黄色，则出现的指示器表示constraints需要被修复。

点击collection view选择它，若看到的依旧是圆圈则再次点击将图标换成绿色**T**型图案，点击并拖动collection view顶部边界的T到名为**Top Layout Guide**的绿色矩形，释放鼠标创建一个与视图上边界关联的constraint。

最后，点击并拖动collection view左侧的T到左边界直接出现蓝色虚线，释放鼠标创建一个与视图左边界关联的constraint。

现在，你的constraints设置应像下面这样:

设置Collection View Cell

也许你注意到了collection view中的包含一个红色感叹号的正方形轮廓，这就是一个colletion view cell，表示collection view中的单个条目。

来设置这个**cell**的尺寸，选择collection view，滚动到**Layout**选项卡的顶部，将**Cell Size**下的**Width**和**Height**设为**100**。

接着点击cell上的红色圆圈，此时会弹出一个提示说你还未设置cell的重用修饰符，选择cell到Widget选项卡中，滚动到底部的**Collection Reusable View**，在**Identifier**中输入**ImageCellIdentifier**，这样错误提示就会消失了。

继续向下滚动到**Interaction Section**，选择**Predefined**(预设)中的blue作为背景色。

这样scene应如下所示:

滚动到**Widget**顶部将**Class**设为**PhotoCollectionImageCell**。

Visual Studio会使用这个名称自动创建一个继承自**UICollectionViewCell**的类，并生成**PhotoCollectionImageCell.cs**文件。

创建Collection View数据源

你需要手动创建一个类，作为 `UICollectionViewDataSource` 为collection view提供数据。

右击 `Solution Explorer` 中的 `ImageLocation` ，选择Add/Class，命名为 `PhotoCollectionDataSource.cs` 点击 `Add` 。

打开新建的 `PhotoCollectionDataSource.cs` ，在文件顶部添加如下代码:

```
using UIKit;
```

这允许你访问iOS的 `UIKit` 框架。

如下改变类的定义:

```
public class PhotoCollectionDataSource : UICollectionViewDataSource
{
}
```

还记得之前定义过collection view cell的重用标识符吗？在这个类中会使用它,在类中进行如下定义:

```
private static readonly string photoCellIdentifier = "ImageCellIdentifier";
```

`UICollectionViewDataSource` 类中包含两个需要你实现的抽象成员。在类中添加如下代码:

```
public override UICollectionViewCell GetCell(UICollectionView collectionView,
    NSIndexPath indexPath)
{
    var imageCell = collectionView.DequeueReusableCell(photoCellIdentifier, indexPath)
       as PhotoCollectionImageCell;

    return imageCell;
}

public override nint GetItemsCount(UICollectionView collectionView, nint section)
{
    return 7;
}
```

`GetCell()` 负责提供collection view中显示的cell。

`DequeueReusableCell` 重用那些不再需要的cell，比如说超出了屏幕。若没有可用的重用cell，会自动创建一个新的cell。

`GetItemsCount` 告诉collection view需要显示7个条目。

接着需要添加一个collection view与ViewContrller类的关联，这样view contrller。回到 `Main.storyboard` ，选择collection view，然后选择 `Widget` 选项卡，在 `Name` 栏输入collectionView。

Visual Studio会使用这个名字自动在ViewContrller类中创建一个静态变量。

打开 `ViewController.cs` ，在类中添加如下代码:

```
private PhotoCollectionDataSource photoDataSource;
```

在 `ViewDidLoad()` 底部，添加如下代码用来初始化数据源并与collection view连接。

```
photoDataSource = new PhotoCollectionDataSource();
collectionView.DataSource = photoDataSource;
```

这样的话 `photoDataSource` 就会给collection view提供数据了。

构建并运行，应该能在collection view中看到七个蓝色方块。

棒 – 有点app的感觉啦!

显示照片

即便蓝色方块比较酷，接着来更新数据源，检索设备中的照片将其显示在collection view中。需要使用Photos框架来访问Photos app所管理的照片与视频资源。

开始，添加在cell上添加一个用来显示图像的视图。打开Main.storyboard选择collection view cell。在Widget选项卡中向下滚动，把背景色改为默认。

打开Toolbox，搜索Image View,，然后把一个Image View拖拽到collection view Cell上。

初始的image view大小会比cell大很多，来重新设定尺寸，选择image view来到Properties\Layout选项卡中。在View区域将X与Y设为0，Width与Height设为100。

切换到image view的Widget选项卡，将Name设为cellImageView。Visual Studio会自动创建一个名为cellImageView的字段。

滚动到**View**区域，将**Mode**改为**Aspect Fill**，这会使图像拉伸以适应边界尺寸。

这个字段并不是public的，其它类无法访问它。需要提供一个方法来设置图像。

打开PhotoCollectionImageCell.cs在类中添加如下方法:

```
public void SetImage(UIImage image)
{
    cellImageView.Image = image;
}
```

现在来检索照片以更新**PhotoCollectionDataSource**。

在**PhotoCollectionDataSource.cs**顶部添加如下行:

```
using Photos;
```

给**PhotoCollectionDataSource**添加如下字段:

```
private PHFetchResult imageFetchResult;
private PHImageManager imageManager;
```

imageFetchResult字段会保存一个照片实体对象们的有序列表，从**imageManager**来得到照片列表。

在**GetCell()**方法上面添加这个构造函数:

```
public PhotoCollectionDataSource()
{
    imageFetchResult = PHAsset.FetchAssets(PHAssetMediaType.Image, null);
    imageManager = new PHImageManager();
}
```

这个构造函数得到了Photos app中的一个所有图像资源的列表，将结果储存进了imageFetchResult字段中。接着设置imageManager，给app用来查询每幅图像的详细信息之用。

在构造函数下面添加析构函数释放imageManager对象。

```
~PhotoCollectionDataSource()
{
    imageManager.Dispose();
}
```

为了让GetItemsCount与GetCell方法使用这些资源并返回images而不是空的cells，如下修改GetItemsCount():

```
public override nint GetItemsCount(UICollectionView collectionView, nint section)
{
    return imageFetchResult.Count;
}
```

如下修改GetCell():

```
public override UICollectionViewCell GetCell(UICollectionView collectionView,
    NSIndexPath indexPath)
{
    var imageCell = collectionView.DequeueReusableCell(photoCellIdentifier, indexPath)

        as PhotoCollectionImageCell;

    // 1
    var imageAsset = imageFetchResult[indexPath.Item] as PHAsset;

    // 2w
    imageManager.RequestImageForAsset(imageAsset,
        new CoreGraphics.CGSize(100.0, 100.0), PHImageContentMode.AspectFill,
        new PHImageRequestOptions(),
         // 3
         (UIImage image, NSDictionary info) =>
        {
           // 4
           imageCell.SetImage(image);
        });

    return imageCell;
}
```

这里是上面方法中变化部分的分析:

1. **indexPath**包含一个collection view所返回item的引用值。这个Item属性是一个索引值，通过索引得到资源并将其转换为**PHAsset**类型。
2. 使用**imageManager**通过期望的尺寸与内容模式请求得到资源图像。
3. 许多iOS框架的方法对于请求使用延迟执行来使它们有足够的时间去完成，比如**RequestImageForAsset**，在完成时会调用一个委托方法。。当请求完成时，会调用使

用image与information的委托方法。
4. 最后将图像设置在cell中。

构建并运行，会看到一个请求访问权限的提示。

选择**OK**，然而app...没反应。太令人失望了!

iOS考虑到访问的用户照片是敏感信息，会提示用户是否给予权限。app必须注册以在用户授权时获得通知，这样就可以重载视图了。下面来做这些工作。

注册监听照片权限的变化

首先需要在PhotoCollectionDataSource类中添加一个方法来通知它若发生变化则重载照片，在类的底部添加如下代码:

```
public void ReloadPhotos()
{
    imageFetchResult = PHAsset.FetchAssets(PHAssetMediaType.Image, null);
}
```

接着，打开**ViewController.cs**在文件顶部引入框架:

```
using Photos;
```

在**ViewDidLoad()**底部添加如下代码:

```
// 1
PHPhotoLibrary.SharedPhotoLibrary.RegisterChangeObserver((changeObserver) =>
{
    //2
    InvokeOnMainThread(() =>
    {
        // 3
        photoDataSource.ReloadPhotos();
        collectionView.ReloadData();
    });
});
```

分析下:

1. app在shared photo library上注册了一个监听者，当photo library变化时会被调用。

2. InvokeOnMainThread()确保UI变化总是在主线程中进行，否则可能会崩溃。
3. 调用**photoDataSource.ReloadPhotos()**来重载照片，调用**collectionView.ReloadData()**来重绘collection view。

最后需要处理最开始的情况——app尚未被允许访问照片，需要请求权限。

在ViewDidLoad()中设置photoDataSource前添加如下代码:

```
if (PHPhotoLibrary.AuthorizationStatus == PHAuthorizationStatus.NotDetermined)
{
    PHPhotoLibrary.RequestAuthorization((PHAuthorizationStatus newStatus) =>
    { });
}
```

检查当前的授权状态，若为NotDetermined则显式地请求访问照片的权限。

为了再次触发照片授权的提示，需要重置iPhone模拟器 **---> Simulator \ Reset Content and Settings**。

构建并运行app，会出现照片授权的提示，选择**OK**后app会在collection view中显示所有设备中照片的缩略图。

最后

完整的Visual Studio项目文件在这里下载

Managing SQLite Database with SwiftyDB

使用SwiftyDB管理SQLite数据库

> - 原文链接 : Managing SQLite Database with SwiftyDB
> - 原文作者 : GABRIEL THEODOROPOULOS
> - 译者 : bugcode

开发应用的过程中总需要一种数据持久化的方案，有很多方式供我们选择:使用本地文件存储，用CoreData或者SQLite数据库。最后一种方式(使用SQLite)有些麻烦，就是在应用程序使用数据库之前首先必须创建好数据库，还要定义好所有数据库表和数据表字段。而且在代码层面来看，使用SQLite管理数据的存储，更新和获取数据并不是那么容易。

上面所说的这些问题貌似被突然出现在GitHub上的一个叫SwiftDB的库给解决了。按作者的话说，这是一个“即引即用”的第三方库。SwiftDB让开发者从手动建库，建表，定义字段的繁杂代码中解脱出来。它会自动提取类中的属性做为数据模型。另外，所有的数据库操作都是对外不可见的(外部看不到具体的数据库操作)，开发者只需要关心应用的业务逻辑。简单而强大的API会让你在处理数据库相关业务的时候倍感舒爽。

有必要提一下的是，你可千万不要以为SwiftDB是万能的。它的确是一个非常靠谱的库，能做的事都做得非常好，但仍然有一些特性不支持，可能在不久的将来这这些特性会被加上。但无论如何，这都是一个非常好的工具库，应该受到重视，所以本入门教程里，我们会介绍SwiftDB库所有的基本功能。

可以在这里找到SwiftDB的文档做参考，当然你得先看完本文再去看文档。可能你已经想要用SQLite了，但是还有些犹豫，那SwiftDB会给你一个定心丸。

上面的说完了，现在我们就一探这个靠谱工具库的究竟。

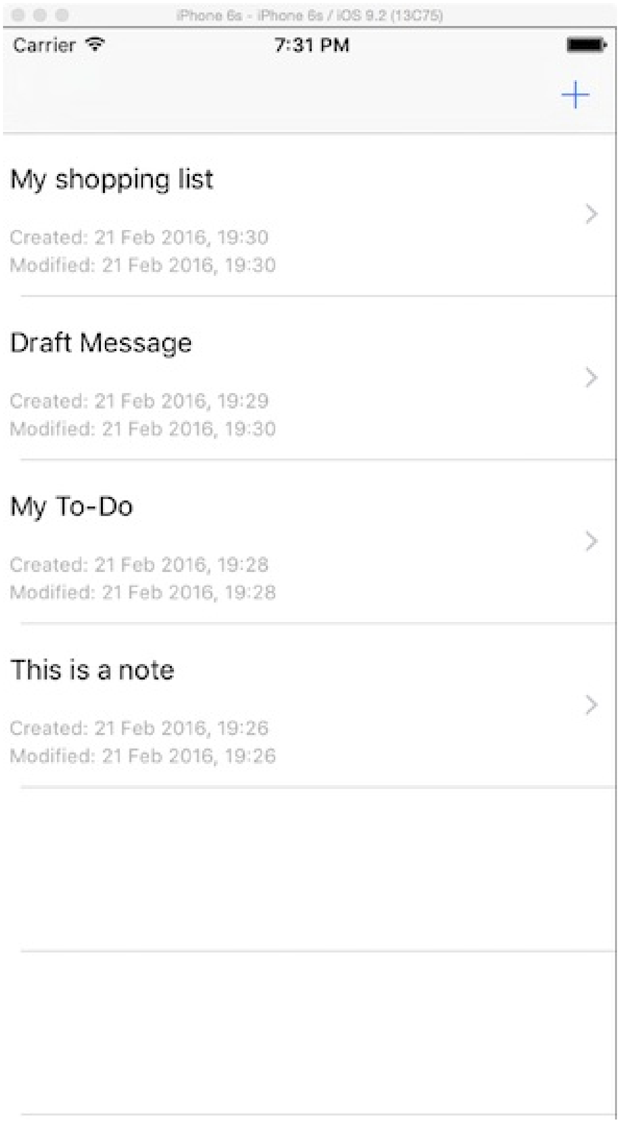
关于Demo

在今天的文章里，我们创建一个简单的笔记应用，能执行所有我们想要的操作:

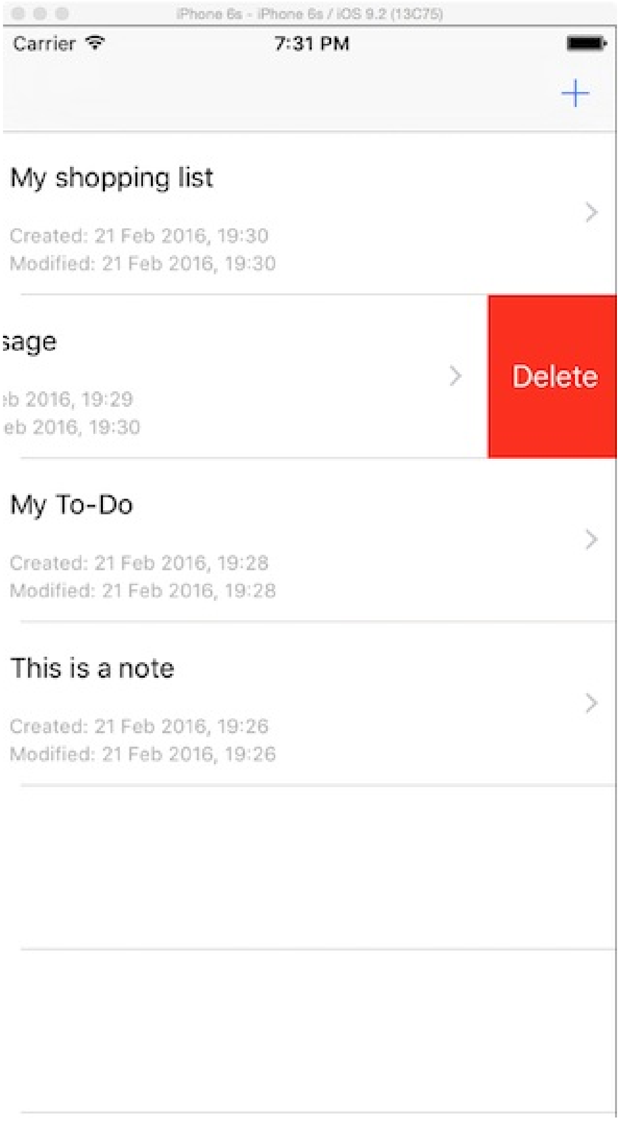
- 笔记列表
- 创建新笔记
- 更新现在笔记
- 删除笔记

显然，SwiftDB会负责管理SQLite数据库里面的数据。所有上面这些列出来的操作足以让你入门SwiftDB。

为了尽可能的保持咱们二边的一致(学习和讲解)，你可以下载这个我已经创建好的初始工程，从这个工程开始学习。下载完之后用Xcode打开浏览一下工程，熟悉一下工程里的内容。如你所见，所有的都是基础的功能，没有数据相关的特性。能跑一下这个工程最好不过了，也好知道app里面大致是什么。 App是基于导航的，在第一个视图控制器里包含了一个展示所有已经保存笔记的tableview:



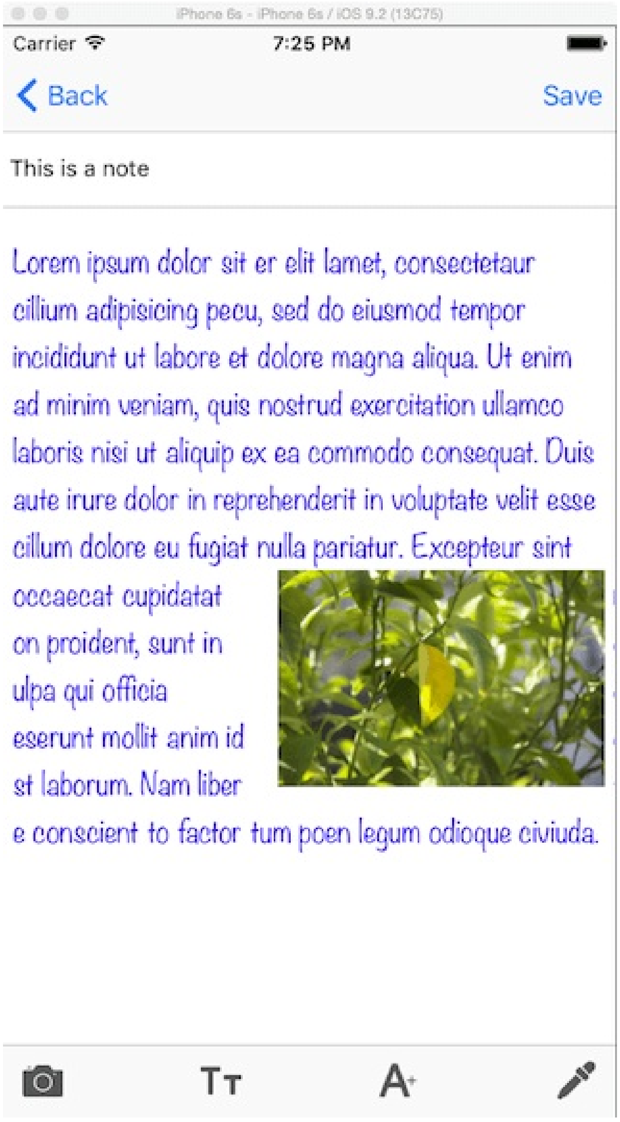
点击每条笔记，可以编辑、修改笔记，我们还可以在每条笔记上左滑来删除笔记。



可以通过点击导航栏上的(+)加号来创建新笔记。为了有足够的操作练习的例子可用，下面列出了在编辑笔记时可用的操作(有已经在demo里实现的，也有新增的)：

1. 设置标题和正文
2. 更改字体
3. 更改字号
4. 更改文本颜色
5. 笔记本引入图片
6. 范围内移动图片，指定图片位置

所有上面列出的操作所产生的数据都会存储在数据库里。为了让上面最后二条更加明晰，我们把真正的图片文件存储在app的文档(documents)目录下，而数据库里只存储对应的图片名称和图片尺寸。不但如此，我们还得创建一个专门管理图片的类(稍后给出具体实现)。



最后还有件重要的事需要说明，即使你已经下载了初始工程，但是在下一节最后的时候，你还是需要一个你自己单独的工作目录。因为我们要用**CocoaPods**来下载SwiftDB库，或者任意其他工程里需要的依赖库。

准备开始吧！但首先，如果你打开了那个初始工程，先关了它。

安装SwiftDB

最先要做的就是下载SwiftDB库，然后放到我们的工程里使用。只是下载库文件再直接加入到工程里这样是不行的，必须使用**CocodPods**依赖管理器来安装SwiftDB库。过程非常简单，即使你之前没用过**CoacoPods**也能马上就搞定。你可以看一下上面给出的CocoaPods链接，参考一下。

安装 CocoaPods

首先要从安装CocoaPods开始。如果已安装过CocoaPods，直接跳过这步。如果没安过，先打开终端(**Terminal**)。输入下面命令来安装CocoaPods:

> sudo gem install cocoapods

回车，输入密码，下载的时候可以稍等一会儿休息下。下载完之后别关终端，一会儿还要用。

安装 SwiftyDB及相关依赖

使用**cd**命令进入到初始工程的根目录下(在终端里):

> cd PATH_TO_THE_STARTER_PROJECT_DIRECTORY

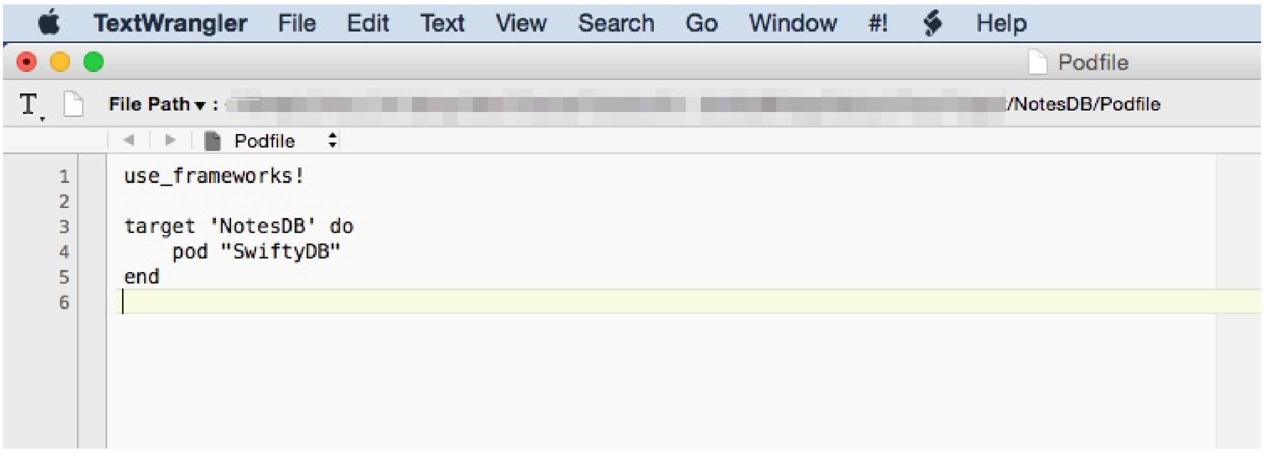
现在该创建**Podfile**了，Podfile文件描述了我们将要下载到CocoaPods里面的库文件。最简单的方式就是就是使用下面的命令让CocoaPods为我们生成一个空Podfile:

> pod init

运行完命令，工程根目录会产生一个名为 `Podfile` 的文件。用编辑器打开(最好别用自带的TextEdit)，然后把下面的内容填入文件中:

```
use_frameworks!

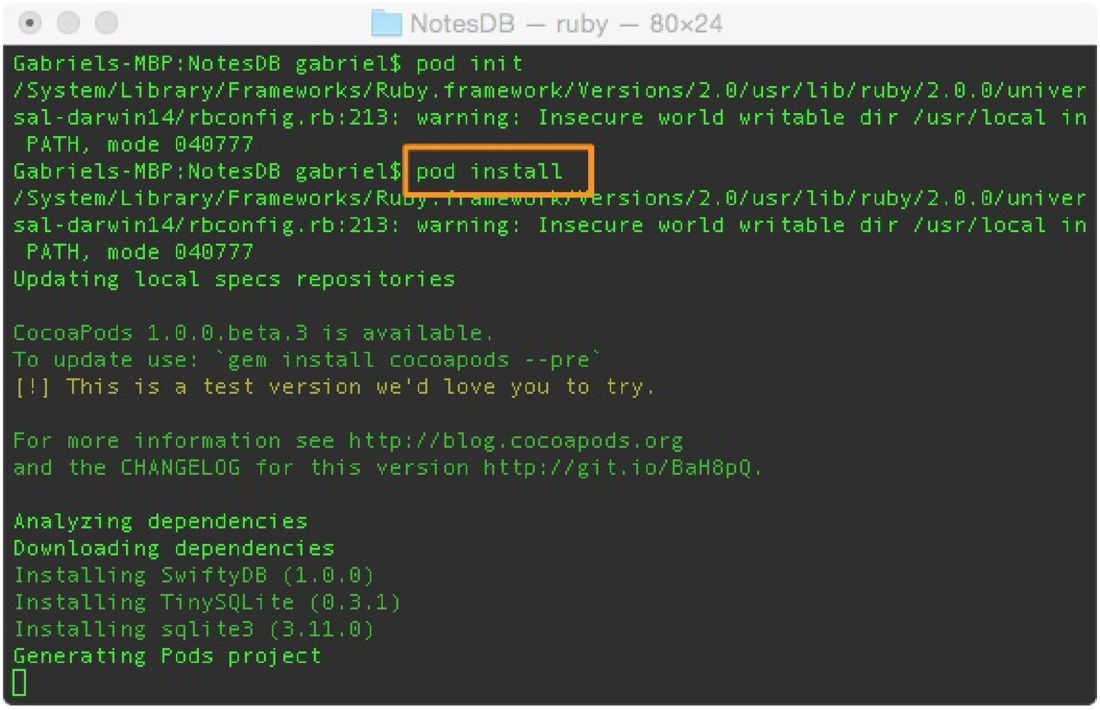
target 'NotesDB' do
    pod "SwiftyDB"
end
```



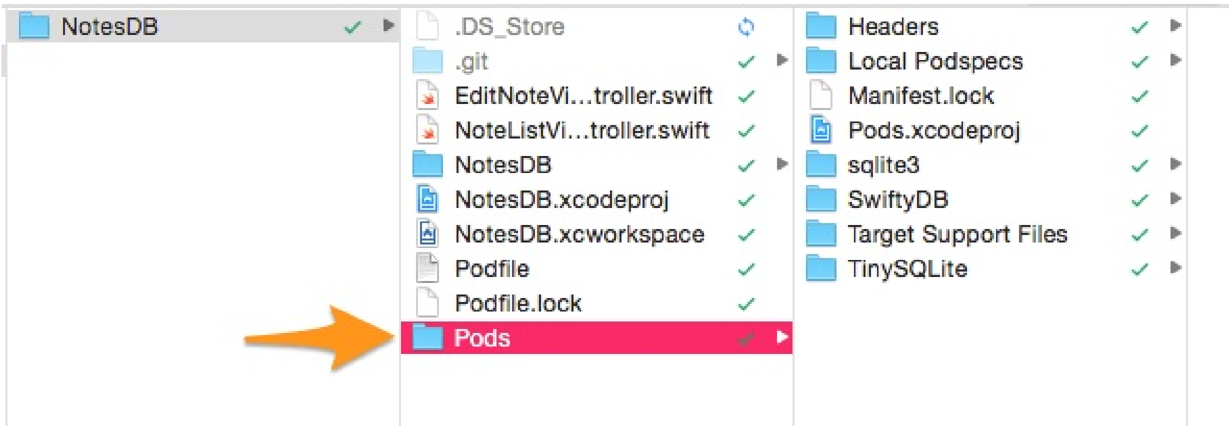
`pod "SwiftDB"` 这一行代码才是真正起作用的。通过那行代码CocoaPods就会下载SwiftDB库和其所有的依赖库，同时还会在Xcode的工作目录中创建多个子目录。

编辑完Podfile，保存关闭。这时确保你已经在Xcode里关了之前打开的初始工程，回到终端输入下面的命令:

> pod install



稍等片刻，然后准备开始吧。这次不要打开之前的初始工程，直接用Xcode打开 `NotesDB.xcworkspace` 。



SwiftyDB起点 – 我们的Model层

在 `NotesDB` 工程里有一个名为 `Note.swift` 的空文件。这个文件就是我们今天的入手点，现在我们要创建几个类用来表示笔记的实体。在理论层面上，接下来要做的就是iOS `MVC` 模式的 `Model` 层的工作。

首先要做的就是引入SwiftDB库，和你猜的一样，在文件的最开头加入如下代码:

```
import SwiftDB
```

现在，定义这个项目中最核心的类:

```
class Note: NSObject, Storable{

}
```

使用SwiftDB的时候有如下二个特殊规则，在上面的类定义中你也能看到:

1. 带有属性的类要用SwiftDB将其存储在数据库中，这个类必须继承 `NSObject` 。
2. 带有属性的类要用SwiftDB将其存储在数据库中，这个类必须适配 `Storable` 协议(是SwiftDB的协议)。

可以考虑在类中添加属性了，针对添加属性SwiftDB有一个新的规则:为了能在数据库中取出数据时加载整个 `Note` 对象，定义的属性 `datatypes` (数据类型)必须是这个列表中已有的类型，不能用简单的数据类型(数组字典)。如果有“不兼容”的数据类型，我们需要使用额外的方法来转换成对应的数据(稍后详述)。SwiftDB默认这样的数据类型是不会被存入数据库的，也不会创建对应的数据表字段。同时，其他任何类中我们想存入数据库的属性我们都需要区别对待。

最后还有一个规则，适配了 `Storable` 协议的的类必须实现 `init` 方法:

```
class Note: NSObject, Storable {

    override required init() {
        super.init()

    }
}
```

现在我们已经有了所有需要的信息，开始定义类属性吧。现在并不定义好所有的属性，因为还有一些属性后期需要讨论一下。所以，这里都是基本的属性:

```
class Note: NSObject, Storable {
    let database: SwiftyDB! = SwiftyDB(databaseName: "notes")
    var noteID: NSNumber!
    var title: String!
    var text: String!
    var textColor: NSData!
    var fontName: String!
    var fontSize: NSNumber!
    var creationDate: NSDate!
    var modificationDate: NSDate!

    ...
}
```

除了第一个，其他不需要额外的注释。第一句代码的作用就是，当对象初始化的时候，如果数据库(名为 `notes.sqlite`)不存在就会创建一个，同时会自动创建一个数据表。数据表的属性与类中定义的属性数据类型相匹配。此外，如果数据库存在，就会直接打开它。。

你可能注意到了，上面我们定义的笔记的各种属性就是描述的要保存到笔记里面的各种元素(标题，正文，文本颜色，字体名字与字号，创建与修改日期)，但是没有任何笔记存储图片的相关信息。是故意这么做的，因为我们需要创建新的图片类，只存储二个属性，图片尺寸和图片名称。

接下来，还是打开 `Notes.swift` 文件，在已有代码的上面或者下面，创建下面的类:

```
class ImageDescriptor: NSObject, NSCoding {
    var frameData: NSData!
    var imageName: String!
}
```

注意表示尺寸的数据类型是用 `NSData` 而不是 `CGRect` 来表示的。这样做很有必要，因为后期向数据库存储数据的时候就相对简单些。对于定义的类要适配 `NSCoding` 协议这个事，在你明白如何做的同时，也要明白为什么要这样做。

回到 `Note` 类，如下所示，在类中定义一个 `ImageDescriptor` 数组:

```
class Note: NSObject, Storable {
    ...

    var images: [ImageDescriptor]!

    ...
}
```

但是现在有一个问题就是，我们应该怎么把 `ImageDescriptor` 对象表示的 `image` 图片数组转换成 `imagesData` 的 `NSData` 对象？方法就是是使用 `NSKeyedArchiver` 类把 `images` 数组打包成一个 `NSDate` 对象。之后会给出详细的实现代码，现在我们需要继续回去写 `ImageDescripter` 类中其他的代码。

如你所知，可以被打包归档(其他语言也叫 `序列化`)的类，其类中的所有属性也必须是可序列化的，我们定义的类满足这个条件， `ImageDescripter` 类中定义的二种数据类型(`NSData` 和 `String`)都可以被序列化。但这还不够，还要对其进行 `encode(编码)` 和 `decode(解码)` ，以便能在合适的时候能打包和解包，这就是为什么类需要适配 `NSCoding` 协议。通过这个协议我们实现下面的几个方法(其中一个是 `init` 方法)，就可以正确的编码和解码类中的二个属性了:

```
class ImageDescriptor: NSObject, NSCoding {
    ...

    required init?(coder aDecoder: NSCoder) {
        frameData = aDecoder.decodeObjectForKey("frameData") as! NSData
        imageName = aDecoder.decodeObjectForKey("imageName") as! String
    }

    func encodeWithCoder(aCoder: NSCoder) {
        aCoder.encodeObject(frameData, forKey: "frameData")
        aCoder.encodeObject(imageName, forKey: "imageName")
    }
}
```

想更多了解 `NSCoding` 协议和 `NSKeydArchiver` 类，可以看下这里和这里，在这里对此不做过多讨论。

除了上边那些，再定义一个便捷的 `init` 方法。方法很简单，不再解释:

```
class ImageDescriptor: NSObject, NSCoding {
    ...

    init(frameData: NSData!, imageName: String!) {
        super.init()
        self.frameData = frameData
        self.imageName = imageName
    }
}
```

到此，我们和SwiftDB打完了第一个照面，即使没做太多的SwiftDB相关工作，但这部分也非常重要，三点原因:

1. 创建一个需要使用SwiftDB功能的类。
2. 了解使用SwiftDB时的一些规则。
3. 知道更多关于使用SwiftDB时对存入数据库的数据类型的限制。

`提示`: 如果现在在Xcode中有些错误，重新构建(Command - B)一次工程错误就没了。

主键和忽略属性

处理数据库的时候推荐使用 `primary keys(主键)` ，这些键值能帮你唯一确定一条数据表中的记录，同时各种操作也依赖主键(如更新一条特定的数据记录)。可以在这里查阅主键的含义。

在SwiftDB中用类中的一个或者多个属性为数据表指定主键非常简单。SwiftDB提供了 `PrimaryKeys` 协议，所有想要在数据库里各数据表中用主键标识唯一类对象的类都可以实现这个协议。

方法非常简单而且很标准，下面直接开始:

在 `NotesDB` 工程里找到 `Extensions.swift` 文件，工程导航栏中点击打开这个文件，加入下面的代码:

```
extension Note: PrimaryKeys {
    class func primaryKeys() -> Set<String> {
        return ["noteID"]
    }
}
```

在这个demo里，我们把 `noteID` 做为sqlite数据库中各数据里唯一的主键。如果需要更多的主键，只需要写在一行用逗号隔开即可。

还有，并不是所有的属性都需要存储在数据库里，需要显式的告知SwiftDB不要包含那些属性。例如，在 `Note` 类中有二个属性不需要存储在数据库中(可能是属性可能存储在数据中，也可能是我们不想存入数据库): `images` 数组和 `database` 数据库实例对象。如何显式排除这二个属性？通过实现另一个SwiftDB提供的叫 `IgnoredProperties` 的协议来做:

```
extension Note: IgnoredProperties {
    class func ignoredProperties() -> Set<String> {
        return ["images", "database"]
    }
}
```

如果还有更多的属性不想存入数据库，只需要把属性加入上面的列表即可。例如，还有下面这个属性:

```
var noteAuthor:String!
```

我们不想把这个属性存入数据库，只需要把这个属性加进 `IgnoredProperties` 协议实现里即可:

```
extension Note: IgnoredProperties {
    class func ignoredProperties() -> Set<String> {
        return ["images", "database", "noteAuthor"]
    }
}
```

`提示` :如果发现有错误，把 `MARKDOWN_HASH6211c316cc840902a4df44c828a26fbeMARKDOWN_HASH` 库引入到 `MARKDOWN_HASH1dbda56f2122b1744ebf59bb64bbffdfMARKDOWN_HASH` 文件中。

保存新笔记

现在已经完成了 `Note` 类的最基本实现，该着手demo应用的功能实现了。目前在新定义的类中没添加任何方法;我们按缺少的功能一步一步的来实现。

首先要有笔记，因此，app需要知道如何用SwiftDB和我们新建的二个类来保存笔记。这些主要都是在 `EditNoteViewController` 类中实现的，在工程导航里打开对应的文件。写代码之前，我把我认为很重要的属性都列在这:

- `imageView` :是个数组，保存了放入笔记中的图片。不要忘了这个数组;之后有了这个数组写代码就便捷多了。
- `currentFontName` :保存当前文本区使用的字体名称。
- `currentFontSize` :当前文本区字体大小。
- `editNoteID` : `noteID` (主键)，用于更新笔记，之后会用到。

由于公用的功能已经在初始工程里写好了，所以下面就是来实现里面缺失的 `saveNote` 方法了。先做两件事: 第一，笔记如果没有标题或者正文不允许保存。第二，忽略保存笔记的时候弹出的键盘。

```
func saveNote() {
    if txtTitle.text?.characters.count == 0 || tvNote.text.characters.count == 0 {
        return
    }

    if tvNote.isFirstResponder() {
        tvNote.resignFirstResponder()
    }
}
```

继续，初始化一个新 `Note` 对象，并为对象内属性赋值。图片需要单独处理，之后会马上处理。

```
func saveNote() {
    ...

    let note = Note()
    note.noteID = Int(NSDate().timeIntervalSince1970)
    note.creationDate = NSDate()
    note.title = txtTitle.text
    note.text = tvNote.text!
    note.textColor = NSKeyedArchiver.archivedDataWithRootObject(tvNote.textColor!)
    note.fontName = tvNote.font?.fontName
    note.fontSize = tvNote.font?.pointSize
    note.modificationDate = NSDate()
}
```

一些注释:

- `noteID` 属性期望一个任意的整型数字来做为主键。你可以创建或者生成任意的整型值，要足够长确保唯一。这里我们直接使用当前时间戳的一部分做为主键，但是在真正的应用程序里这样做并不是很好的方法，因为时间戳里包含了太多的数字。在我们的demo程序里还好，只是简单的用int值。
- 当我们第一次保存笔记的时候，同时设置笔记创建时间和修改时间为当前时间戳(用 `NSDate` 对象表示)。
- 唯一需要我们做一下特殊的转换的就是，从 `NSData` 对象中取出当前文本区的文本颜色。文本颜色的对象值是通过 `NSKeyedArchiver` 类打包过的。

现在重点看一下怎样保存图片。我们需要写一个新方法来填充图片数组。方法里做二个事情:保存真正的图片到应用程序的文稿(documents)目录中，同时为每个图片创建相应的 `ImageDecripter` 对象。这些对象都会被放在 `images` 数组里。

为创建这个新的方法，我们需要迂回一下，再次回到 `Note.swift` 文件。先看具体实现，后面再具体讨论细节。(原文排版有误)

```
func storeNoteImagesFromImageViews(imageViews: [PanningImageView]) {
    if imageViews.count > 0 {
        if images == nil {
            images = ImageDescriptor
        } else {
            images.removeAll()
        }


        for i in 0..<imageViews.count {
            let imageView = imageViews[i]
             let imageName = "img_\(Int(NSDate().timeIntervalSince1970))_\(i)"


            images.append(ImageDescriptor(frameData: imageView.frame.toNSData(), image
Name: imageName))

            Helper.saveImage(imageView.image!, withName: imageName)
        }

        imagesData = NSKeyedArchiver.archivedDataWithRootObject(images)
    } else {
        imagesData = NSKeyedArchiver.archivedDataWithRootObject(NSNull())
}
```

这里解释为什么要写上面这个方法:

1. 最开始检查 `images` 数组是否已经初始化。如果为空，初始化数组，非空则移除内部所有数据。第二步在我们更新一个已存在的笔记的时候非常有用。

2. 针对每个图片创建唯一的名称。图片名称类似于:“img_12345679_1”。
3. 传入图片尺寸和图片名称参数给我们自定义的构造方法来初始化一个 `ImageDescripter` 对象。 `toNSData()` 方法已经做为 `CGRect` 的扩展实现，可以 `Extensions.swift` 中找到。作用就是转换尺寸为 `NSData` 对象。 `ImageDecripter` 初始化完成后，直接放入 `images` 数组中。
4. 保存真正的图片到应用程序的文稿目录中。 `saveImage(_:withName:)` 类方法在 `Helper.swift` 中实现，那里还有更多有用的方法。
5. 最后所有的图片都处理完成，就把 `images` 数组通过打包转换成 `NSData` 对象，并赋值给 `imagesData` 属性。最后一行代码才是 `ImageDescripeter` 类适配 `NSCoding` 协议的真正意义所在。

`else` 分支看似无用，其实是必需的。默认情况笔记中加图片时 `imagesData` 就是空值。但是SQLite并不能识别“nil”值是空值。SQLite只认 `NSNull` ，所以我们需要转为 `NSData` 。

回到 `EditNoteViewController.swift` 文件，加入下面的我们刚刚创建的方法:

```
func saveNote() {
    ...

    note.storeNoteImagesFromImageViews(imageViews)
}
```

现在回到 `Note.swift` 文件中，实现真正向数据库中存入笔记数据的方法。此时有些重要的东西需要你了解一下: SwiftDB提供了执行任意关系型数据库的同步或者异步操作的选项。使用哪种方式取决于你的app的类型。我建议使用异步的方式，因为这样在处理数据库操作的时候不会对主线程加锁，也不会锁定UI而影响用户体验。但是重申一下，用什么方法，这一切都取决于你自己。

这里我们使用异步的方式把笔记数据保存到数据库。SwiftDB里的方法包含了一个返回操作结果的闭包。返回的结果对象的更详细信息可以看这里)，建议你现在就看一下。

现在看看我们一直在讨论的这个新方法:

```
func saveNote(shouldUpdate: Bool = false, completionHandler: (success: Bool) -> Void)
{
    database.asyncAddObject(self, update: shouldUpdate) { (result) -> Void in
        if let error = result.error {
            print(error)
            completionHandler(success: false)
        }
        else {
            completionHandler(success: true)
        }
    }
}
```

上面的代码很简单，这个方法也用来更新笔记。通过事先设置 `shouldUpdate` 这个布尔值，`asyncDataObject(...)` 方法根据这个值来确定是创建一个新的数据库记录还是更新已有的笔记记录。

还有，你看方法中第二个参数，是完成时的处理函数。我需要用合适的参数在笔记是否保存完成后调用这个方法。建议你在后台有异步的任务执行的时候一直用完成回调处理函数。这样的话，在你后台任务执行完成的时候，你就能通知调用者的函数，那时就可以做任意的数据转换，或者数据备份。

上面你所看到的在其他关系型数据库也能看到。我们应该总是对结果进行错误检查，无论是否有错误我们都应该检查。上面的例子如果发生错误，调用完成处理函数的时候就需要传 `false` 值，意思就是笔记保存失败。反之就是传 `true` 表示保存操作成功执行。

再次回到 `EditNoteViewController` 类，完成 `saveNote()` 方法。我们调用一下刚写好的方法，如果保存成功则弹出一个窗口，如果发生错误就显示一条错误信息。

```
func saveNote() {
    ...

    let shouldUpdate = (editedNoteID == nil) ? false : true

    note.saveNote(shouldUpdate) { (success) -> Void in
        dispatch_async(dispatch_get_main_queue(), { () -> Void in
            if success {
                self.navigationController?.popViewControllerAnimated(true)
            }
            else {
                let alertController = UIAlertController(title: "NotesDB", message: "An
 error occurred and the note could not be saved.", preferredStyle: UIAlertControllerSt
yle.Alert)
                alertController.addAction(UIAlertAction(title: "OK", style: UIAlertAct
ionStyle.Default, handler: { (action) -> Void in

                }))
                self.presentViewController(alertController, animated: true, completion
: nil)
            }
        })
    }
}
```

注意上面实现中的 `shouldUpdate` 变量。由 `editedNoteID` 是否为空来决定其值。表示笔记是否是被更新。

这时候你可以跑一下程序并试着保存一下笔记。如果你是按教程一步一步做到这步，那保存笔记应该不会出任何问题。

加载笔记列表

创建和保存新笔记的功能已经能用了，现在可以开始实现从数据库加载已保存的笔记的功能了。加载完的笔记就意味着要 `NoteListViewController` 类中展示出来。但是开始实现这个类之前，我们先在`Note.swift文件中新创建一个从数据库中加载数据的方法。

```
func loadAllNotes(completionHandler: (notes: [Note]!) -> Void) {
    database.asyncObjectsForType(Note.self) { (result) -> Void in
        if let notes = result.value {
            completionHandler(notes: notes)
        }

        if let error = result.error {
            print(error)
            completionHandler(notes: nil)
        }
    }
}
```

真正执行加载数据的操作是SwiftDB库的 `asyncObjectsForType(...)` 方法，这个方法是异步的。方法返回结果可能包含错误，也可能是一个从数据库中加载出来的笔记对象的集合(数组)。第一种情况，从数据库加载数据时候出错，我们调用加载完成回调并传给其nil参数，告知调用者加载数据出现了问题。第二种情况，我们把 `Note` 对象集合传给加载完成的回调函数，这样我们就可以在方法的外部使用加载出来的笔记对象数组了。

现在我们打开 `NoteListViewController.swift` 文件。首先得定义一个包含了 `Note` 对象(对象都是从数据库中获取的)的数组。这个数组将做为tableview的datasource(很显然的事情)。那么就在类的开头加入如下代码:

```
var notes = [Note]()
```

另外还要初始化一个新的 `Note` 对象，这样我们才能调用先前创建的 `loadAllNotes(...)` 方法。

```
var note = Note()
```

现在就写一个特别简单的方法，调用我们上面刚写完的方法，从数据库中获取所有存储的笔记数据并放到 `notes` 数组中。

```
func loadNotes() {
    note.loadAllNotes { (notes) -> Void in
        dispatch_async(dispatch_get_main_queue(), { () -> Void in
            if notes != nil {
                self.notes = notes
                self.tblNotes.reloadData()
            }
        })
    }
}
```

注意，当加载完所有的笔记数据之后，我们要在主线程里重新加载tableview。在闭包里我们把数据都保存在 `notes` 数组中了。

通过上面二个方法，我们就可以获取所有数据库中存储的笔记数据了。很简单吧！但不要忘了在某个地方调用 `loadNotes()` 方法，这里我们把它写在 `viewDidLoad()` 方法里:

```
override func viewDidLoad() {
    ...

    loadNotes()
}
```

只加载数据还不够，加载完成我们还要用上这些数据。现在我们改一下tableview的相关方法，从tableview的行数开始:

```
func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return notes.count
}
```

接下来，在tableview上显示一些笔记的数据。具体来说，就是要显示每个笔记的标题，创建时间和修改时间:

```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -
> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("idCellNote", forIndexPath:
 indexPath) as! NoteCell

    let currentNote = notes[indexPath.row]

    cell.lblTitle.text = currentNote.title!
    cell.lblCreatedDate.text = "Created: \(Helper.convertTimestampToDateString(current
Note.creationDate!))"
    cell.lblModifiedDate.text = "Modified: \(Helper.convertTimestampToDateString(curre
ntNote.modificationDate!))"

    return cell

}
```

现在跑一下程序，你创建的所有的笔记都将显示在tableview上了。

另一种获取数据的方式

之前我们用SwiftDB库提供的 `asyncObjectsForType(...)` 方法从数据库中获取笔记数据。方法返回一个对象的数组(我们的例子是 `Note` 对象)，我感觉这非常简便。但这种从数据库中获取对象数据的方法也不是万能的；有些情况下直接获取原始数据值数组的方式可能更简便一些。

SwiftDB已经替你想到了，它提供了另一种方式从数据库中获取数据。有一个叫 `asyncDataForType(...)` (同步操作就用 `dataForType(...)` 方法)的方法。方法返回这种形式的字典集合: `[[String:SQLiteValue]]` (`SQLiteValue` 表示任意可用的数据类型)。

可以在这里和这里了解更多相关信息。增强一下 `Note` 类做为练习，让其不单单能获取对象，还能加载简单数据类型。

更新笔记

我们的demo程序其中一个功能是编辑并且修改已保存的笔记。也就是说，我们通过点击当前的笔记的cell，选中对应的笔记，然后在 `EditNoteViewController` 中把对应笔记的详情展示出来。同时修改完后还能再次保存到数据库。

打开 `NoteListViewController.swift` 文件，我们需要定义一个新的属性，用来保存已选中笔记的ID，在类的开头加入如下代码:

```
var idOfNoteToEdit: Int!
```

现在我们实现下一个 `UITableViewDelegate` 方法，在这个方法里我们根据选中的cell获取笔记的 `noteID` ，获取后马上执行页面跳转，显示 `EditNoteViewController` :

```
func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath)
 {
    idOfNoteToEdit = notes[indexPath.row].noteID as Int
    performSegueWithIdentifier("idSegueEditNote", sender: self)
}
```

在 `prepareForSegue(...)` 方法中，我们把 `idOfNoteToEdit` 值传给之后要显示的视图控制器(view controller):

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if let identifier = segue.identifier {
        if identifier == "idSegueEditNote" {
            let editNoteViewController = segue.destinationViewController as! EditNoteV
iewController

            if idOfNoteToEdit != nil {
                editNoteViewController.editedNoteID = idOfNoteToEdit
                idOfNoteToEdit = nil
            }
        }
    }
}
```

现在这个功能已经完成一半了。在继续开始 `EditNoteViewController` 类的工作之前，为能访问 `Note` 类，我们先来实现另一个简单的函数，通过给定的 `noteID` 获取对应已存储的笔记实例对象。下面是具体实现:

```
func loadSingleNoteWithID(id: Int, completionHandler: (note: Note!) -> Void) {
    database.asyncObjectsForType(Note.self, matchingFilter: Filter.equal("noteID", val
ue: id)) { (result) -> Void in
        if let notes = result.value {
            let singleNote = notes[0]

            if singleNote.imagesData != nil {
                singleNote.images = NSKeyedUnarchiver.unarchiveObjectWithData(singleNo
te.imagesData) as? [ImageDescriptor]
            }

            completionHandler(note: singleNote)
        }

        if let error = result.error {
            print(error)
            completionHandler(note: nil)
        }
    }
}
```

这里有一个新用到的东西，我们通过 `filter` 筛选获取的数据结果。使用Fileter类的 `equal(...)` 类方法来设置过滤我们想要的结果。记得看一下这个链接，有更多的方式来筛选从数据库获取的数据或者对象。

通过上文所示的筛选方式，我们把 `noteID` 传给方法做为其参数，让SwiftDB只加载对应 `noteID` 的数据库记录。当然，我们知道这只会返回一条数据库记录，因为 `noteID` 这个字段是主键，数据库里相同的主键不可能返回多条记录。

获取的返回结果是一个 `Note` 对象数组，所有必须拿到数组里的第一个元素(也只包含一个元素)。然后就是把里面的图片数据转换为 `ImageDescriptor` 对象数组，并赋给 `images` 变量。这步很重要，如果略过这步，数据库里的图片原始数据就不能被正确加载显示。

最后根据是否成功从数据库获取笔记数据来调用完成处理函数。第一种情况，我们成功获取到数据并把其传给调用者让调用者使用。还有第二种情况，数据获取出错，传nil给调用者，表示没有获取到任意数据。

现在我们可以打开 `EditNoteViewController.swift` 文件了，在类中定义并初始化一个 `Note` 对象属性:

```
var editNote = Note()
```

这个定义好的对象首先会被用于调用我们上文定义好的方法，调用之后，这个对象就包含了从数据库获取的数据。

我们只需要加载通过 `editedNoteID` 属性用 `loadSingleNote(...)` 方法获取的数据。现在需要定义 `viewWillAppear(_:)` 方法，在这个方法里再写详细的逻辑。

在下面的代码段里你会看到，当通过 `loadSingleNoteWithID(...)` 方法中的完成回调函数获取的笔记数据时，在完成回调里所有的对应属性都会被填充。我们可以通过这些设置笔记的标题，正文，文本颜色，字体等一系列属性。如果笔记中带有图片，还得使用 `ImageDescriptor` 中的尺寸数据为每个图片创建image视图。

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    if editedNoteID != nil {
        editedNote.loadSingleNoteWithID(editedNoteID, completionHandler: { (note) -> V
oid in
            dispatch_async(dispatch_get_main_queue(), { () -> Void in
                if note != nil {
                    self.txtTitle.text = note.title!
                    self.tvNote.text = note.text!
                    self.tvNote.textColor = NSKeyedUnarchiver.unarchiveObjectWithData(
note.textColor!) as? UIColor
                    self.tvNote.font = UIFont(name: note.fontName!, size: note.fontSiz
e as CGFloat)

                    if let images = note.images {
                        for image in images {
                            let imageView = PanningImageView(frame: image.frameData.to
CGRect())

                            imageView.image = Helper.loadNoteImageWithName(image.image
Name)

                            imageView.delegate = self
                            self.tvNote.addSubview(imageView)
                            self.imageViews.append(imageView)
                            self.setExclusionPathForImageView(imageView)
                        }
                    }

                    self.editedNote = note

                    self.currentFontName = note.fontName!
                    self.currentFontSize = note.fontSize as CGFloat
                }
            })
        })
    }
}
```

把所有值都填充完别忘了把 `note` 对象赋值给 `editedNote` 对象，后面我们还会用到。

还有最后一个必要的步骤：修改 `saveNote()` 方法，就是当已存在笔记更新的时候，不要再创建一个 `Note` 对象设置主键和创建日期，不要把它当做一个新笔记再次保存。

在 `saveNote()` 方法里找到下面这三行:

```
let note = Note()
note.noteID = Int(NSDate().timeIntervalSince1970)
note.creationDate = NSDate()
```

替换为下面的代码:

```
let note = (editedNoteID == nil) ? Note() : editedNote

if editedNoteID == nil {
    note.noteID = Int(NSDate().timeIntervalSince1970)
    note.creationDate = NSDate()
}
```

其余方法不变(至少目前不变)。

更新笔记列表

如果你现在测试一下，你会发现创建新笔记或者更新已存在的笔记时，笔记列表不会更新。这是正常的，因为程序里还没有实现这个功能，这部分内容里我们就来修复这个非预期的行为。

正如你猜的，我们使用 `代理模式(Delegation pattern)` 在 `EditNoteViewController` 类中修改笔记的时候通知 `NoteListViewController` 类。入手点就是在 `EditNoteViewController` 里创建一个新协议，里面有二个必须实现的方法，如下:

```
protocol EditNoteViewControllerDelegate {
    func didCreateNewNote(noteID: Int)

    func didUpdateNote(noteID: Int)
}
```

这二个代理方法都需要一个笔记ID的参数，表示新创建的或者已编辑更新的笔记的ID。现在，在 `EditNoteViewController` 类中加如下属性:

```
var delegate: EditNoteViewControllerDelegate!
```

最后再次回到之前的 `saveNote()` 方法中。在完成回调函数中定位到下面这行代码:

```
self.navigationController?.popViewControllerAnimated(true)
```

用下面这段代码替换这行代码:

```
if self.delegate != nil {
    if !shouldUpdate {
        self.delegate.didCreateNewNote(note.noteID as Int)
    }
    else {
        self.delegate.didUpdateNote(self.editedNoteID)
    }
}
self.navigationController?.popViewControllerAnimated(true)
```

至此，之后每次创建新笔记或者编辑更新笔记都会调用对应的代理方法。但是到现在为止工作只做了一半。回到 `NoteListViewController.swift` 文件，首先在类开头适配我们刚才创建的新协议:

```
class NoteListViewController: UIViewController, UITableViewDelegate, UITableViewDataSo
urce, EditNoteViewControllerDelegate {
   ...
}
```

接下来，在 `prepareForSegue(...)` 方法中设置 `EditNoteViewController` 做为这个类的代理。在 `let editNoteViewController = segue.destinationViewController as! EditNoteViewController` 这行代码之后加入如下所示代码:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if let identifier = segue.identifier {
        if identifier == "idSegueEditNote" {
            let editNoteViewController = segue.destinationViewController as! EditNoteV
iewController

            editNoteViewController.delegate = self  // Add this line.

            ...
        }
    }
}
```

非常好，现在大部分工作都差不多做完了。不过我们还忘了实现代理方法。首先处理创建新笔记的情况:

```
func didCreateNewNote(noteID: Int) {
    note.loadSingleNoteWithID(noteID) { (note) -> Void in
        dispatch_async(dispatch_get_main_queue(), { () -> Void in
            if note != nil {
                self.notes.append(note)
                self.tblNotes.reloadData()
            }
        })
    }
}
```

通过 `noteID` 参数，从数据库获取笔记数据(如果数据存在)，直接加到 `notes` 数组中，并重新加载tableview。

下面来看另一种情况:

```
func didUpdateNote(noteID: Int) {
    var indexOfEditedNote: Int!

    for i in 0..<notes.count {
        if notes[i].noteID == noteID {
            indexOfEditedNote = i
            break
        }
    }

    if indexOfEditedNote != nil {
        note.loadSingleNoteWithID(noteID, completionHandler: { (note) -> Void in
            if note != nil {
                self.notes[indexOfEditedNote] = note
                self.tblNotes.reloadData()
            }
        })
    }
}
```

这种情况，我们找到已更新笔记在 `notes` 数组中的索引位置。一旦笔记更新，便从数据库中载入已更新笔记并把旧的笔记用新的替换掉。同时刷新tableievw，新的笔记修改日期和修改过的笔记就会马上显示出来。

删除记录

最后，这个笔记demo还缺一个比较重要的功能，就是删除笔记记录。很显然我们还需要在 `Note` 类中添加最后一个方法。这个方法在每次删除笔记的时候都会调用。打开 `Note.swift` 文件。

这部分里我们唯一需要新加的东西就是用SwiftDB的方法在数据库中执行真正的数据删除操作，也就是下面你将看到的代码实现。像之前其他操作一样，这是多个异步执行的操作，有一个完成时的回调在操作执行完成的时候调用。最后，筛选出需要从数据库中删除的列。

```
func deleteNote(completionHandler: (success: Bool) -> Void) {
    let filter = Filter.equal("noteID", value: noteID)

    database.asyncDeleteObjectsForType(Note.self, matchingFilter: filter) { (result) -
> Void in
        if let deleteOK = result.value {
            completionHandler(success: deleteOK)
        }

        if let error = result.error {
            print(error)
            completionHandler(success: false)
        }
    }
}
```

现在我们打开 `NoteListViewController.swift` ，定义下一个 `UITableViewDataSource` 方法:

```
func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCel
lEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
    if editingStyle == UITableViewCellEditingStyle.Delete {

    }
}
```

在我们已有代码中添加以上方法，这样每次你在笔记记录的cell上左滑的时候，默认的 `Delete` 按钮就会在cell的右侧显示出来。当点击cell右侧出现的删除按钮的时候，就会执行上面定义的方法中 `if` 语句里面的逻辑。现在在其中加入如下代码:

```
func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
    if editingStyle == UITableViewCellEditingStyle.Delete {
        let noteToDelete = notes[indexPath.row]

        noteToDelete.deleteNote({ (success) -> Void in
            dispatch_async(dispatch_get_main_queue(), { () -> Void in
                if success {
                    self.notes.removeAtIndex(indexPath.row)
                    self.tblNotes.reloadData()
                }
            })
        })
    }
}
```

解释一下上面的代码，首先，在保存笔记对象的集合里找到与选中的笔记相一致的笔记对象。接下来，调用我们先前在 `Note` 类中定义的删除方法删除这个笔记。如果操作成功，会把 `notes` 数组中对应的 `Note` 对象移除，同时重新载入tableview刷新UI界面。

就是这样！

怎么排序呢?

可能你想知道如何从数据库取出数据时候就对我们的数据进行排序。排序非常有用，根据一个或者多个字段进行升序或者降序操作，最后返回改变顺序的数据。例如，我们可以对我们保存的笔记按这样的方式进行排序，修改日期越临近现在的位置越靠上。

不幸的是在写这篇入门教程的时候SwiftDB并不支持数据排序。这使用起来确实非常不便利。有一个解决方式:在需要的地方手动对数据进行排序。为了演示这个，我们在 `NoteListViewController.swift` 文件里创建最后一个方法， `sortNotes()` 方法。在这里我们直接用Swift自带的 `sort()` 函数:

```
func sortNotes() {
    notes = notes.sort({ (note1, note2) -> Bool in
        let modificationDate1 = note1.modificationDate.timeIntervalSinceReferenceDate
        let modificationDate2 = note2.modificationDate.timeIntervalSinceReferenceDate

        return modificationDate1 > modificationDate2
    })
}
```

因为 `NSDate` 对象不能直接进行比较，首先我们需要把它转换为时间戳(double类型)。我们执行比较操作之后返回排序结果。上面的代码对笔记排序，修改时间最靠近现在的就会排在 `notes` 数组里第一个位置。

上面定义的方法必须在每次 `notes` 数组发生变化的时候都调用。首先，我们先像下面这样修改 `loadNotes` 方法:

```
func loadNotes() {
    note.loadAllNotes { (notes) -> Void in
        dispatch_async(dispatch_get_main_queue(), { () -> Void in
            if notes != nil {
                self.notes = notes

                self.sortNotes()  // Add this line to sort notes.

                self.tblNotes.reloadData()
            }
        })
    }
}
```

同时也得像下面这样修改二个代理方法:

```
func didCreateNewNote(noteID: Int) {
    note.loadSingleNoteWithID(noteID) { (note) -> Void in
        dispatch_async(dispatch_get_main_queue(), { () -> Void in
            if note != nil {
                self.notes.append(note)

                self.sortNotes() // Add this line to sort notes.

                self.tblNotes.reloadData()
            }
        })
    }
}
```

```
func didUpdateNote(noteID: Int) {
    ...

    if indexOfEditedNote != nil {
        note.loadSingleNoteWithID(noteID, completionHandler: { (note) -> Void in
            if note != nil {
                self.notes[indexOfEditedNote] = note

                self.sortNotes()  // Add this line to sort notes.

                self.tblNotes.reloadData()
            }
        })
    }
}
```

现在再运行一下程序，你就会发现tableview里面的笔记列表已经按笔记修改日期来排列了。

结语

不可否认，SwiftDB是个非常棒的工具，可以不用太大成本就能集成在很多的应用程序里。我们都认同这一点，就是在我们的应用内同时使用几个数据库的时候，SwiftDB做到了快速可靠。在这个demo的入门教程中贯穿了这个库的基本用法，但你还需要去了解更多。当然，有正式的文档供你查阅让你参考。在我们今天这个例子中，出于新手入门的目的，我们创建了只有一个表的数据库，这个表的字段也只匹配 `Note` 类。而在真正的应用程序中，只要你在代码中创建对应的数据模型(不同的类)，就可以按你所想有任意多的数据表。就我个人而言，我是一定会在我的项目中使用SwiftDB的，实际上我已经开始计划了。不管怎样，你现在已经了解SwiftDB了，已经知道它如何工作以及如何使用它了。是不是把它放到你的工具箱里，这由你决定。总之，我希望你阅读这篇入门的时间没有浪费，并希望你在此学会了一二个新东西。在下一个教程发布之前，过得愉快！

你可以在GitHub下载整个工程)做为参考。

Detecting low Power Mode in Swift

使用Swift检测省电模式

> - 原文链接：Detecting low Power Mode in Swift
> - 原文作者：iOS-Blog Admin
> - 译者：bugcode

在本篇入门教程中，会为你展示如何检测用户的设备进入省电模式，并且在设备进入省电模式之后还能保持后台程序刷新。我会教你使用 `NSProcessInfo` 类中的 `LowPowerEnabled` 属性，同时使用 `NSProcessInfoPowerStateDidChangeNotification` 监听省电模式状态的变化(省电模式和非省电模式状态改变)来实现所说的功能。



> 截止目前，Xcode的iOS模拟器没有省电模式的开关选项，所以要测试的话，需要在真机上跑测试工程。

目录

省电模式是什么?

省电模式是苹果处理电池电量不断减少的一种手段。省电模式会停止所有的后台活动，调暗屏幕同时使用各种方式来保存电量。直接且有效，我曾经1%的电量用了半小时，听起来很假，但这是事实。

为什么要检测省电模式

这样一个情景：

假设你用的app在后台每15分钟连接一次web服务。类似iOS的邮件app(连接频度取决于你的设置)。app会更新数据，用户接收到重要的数据的时候希望有提示。

Linda是广大用户中的一位，她通过这些消息通知才能在这个快节奏的社会中赖以生存，她的圈子和工作都需要这些消息通知，但是省电模式开了会完全停掉app的后台刷新。现在好了，Linda得一小时后才会有电源，等到那个时候省电模式才会关闭。一小时过后，她给设备插上电源，为今天松了一口气。之后的一天晚上，她遇见了一个熟知最新八卦、行业新闻的朋友Sarah，Sarah正准备去过周末因为她捡了个大便宜。Linda得知后一脸迷惑，因为她没有得到任何关于此事的消息，错过了本来就快要结束的Ebay促销和29刀的周末团购优惠券。Linda十分沮丧，意识到在省电模式过后app没有还原成之前的设置，也没有进行它往常的后台活动，这一切让她感到糟透了。

不要伤害像Linda这样的人。

现在，你可以检测省电模式的状态，从而可以相应的控制你的app的活动状态。

这篇教程假定你已掌握Swift2

创建检测省电模式的工程

针对这个教程，我们创建一个简单的下载youtube视频的app，实际只做简单的加载并不会下载内容提取到设备上。我们将会使用swift来检测省电模式，同时会延迟省电模式的相关动作直接到省电模式被停用。

创建新的单页应用，任意取名并放在合适的位置。

首先需要在代码前边定义一些变量，在ViewController.swift文件中的类定义里这样写：

```
var StartVideoDownload = true
let processInfo = NSProcessInfo.processInfo()
```

> `NSProcessInfo`对象会在这个方法第一次调用的时候被创建，之后每次的调用都只会返回这个已经被创建好的对象。

现在写一个方法用来下载:

```
func downloadStart(){
    guard let url = NSURL(string: "https://www.youtube.com/embed/YQHsXMglC9A") where
       !processInfo.lowPowerModeEnabled else{
       return
    }
StartVideoDownload = false
}
```

这个要下载的视频是Adelle的Hello。不要吐槽我。

现在我们要实现 `powerModeChanged(_:)` 方法。

```
func powerModeChanged(notif: NSNotification){
     guard mustDownloadVideo else{
         return
     }
downloadStart()
}
```

这就是在app里检测到省电模式并继续为用户提供下载的一个简单解决方案。

Designing Animations with UIViewPropertyAnimator in iOS 10 and Swift 3

在iOS 10中使用Swift 3与UIViewPropertyAnimator设计动画

> - 原文链接：Designing Animations with UIViewPropertyAnimator in iOS 10 and Swift 3
> - 原文作者：Jason Newell
> - 译者：yrq110

---

iOS 10中的UIKit中加入了一些新的对象与协议，具有“基于对象、完全交互，支持可中断动画来保持对动画的控制，使用基于手势的交互来连接”等特点。

简而言之就是为了给开发者可扩展的、更上层的控制，比如一些时间函数，更方便地去取消、回放、暂停与重启动画, and change the timing and duration on the fly with smoothly interpolated results. 这些动画的功能同样适用于view controller的变换。

我会简单的介绍一些基本用法，并且关注一些文档与讨论中未涉及的关键点。

构建基础App

待会儿我们要尝试一些UIViewPropertyAnimator的特性，不过首先需要一些动画。

创建一个单视图应用，在ViewContrller.swift中添加如下代码。

```
import UIKit

class ViewController: UIViewController {
    // this records our circle's center for use as an offset while dragging
    var circleCenter: CGPoint!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Add a draggable view
        let circle = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 100.0, height: 100.0))
        circle.center = self.view.center
        circle.layer.cornerRadius = 50.0
        circle.backgroundColor = UIColor.green()

        // add pan gesture recognizer to
        circle.addGestureRecognizer(UIPanGestureRecognizer(target: self, action: #selector(self.dragCircle)))

        self.view.addSubview(circle)
    }

    func dragCircle(gesture: UIPanGestureRecognizer) {
        let target = gesture.view!

        switch gesture.state {
        case .began, .ended:
            circleCenter = target.center
        case .changed:
            let translation = gesture.translation(in: self.view)
            target.center = CGPoint(x: circleCenter!.x + translation.x, y: circleCenter!.y + translation.y)
        default: break
        }
    }
}
```

这儿没啥难的。在viewDidLoad中创建一个绿色的圆将其定位在屏幕中央，之后绑定一个UIPanGestureRecognizer手势，使dragCircle可以响应在它身上的拖动事件:在屏幕上移动圆圈。如你所想，结果就是一个可拖拽的圆:

关于UIViewPropertyAnimator

我们主要使用UIViewPropertyAnimator类给view属性添加动画效果、中断动画，与改变动画进行过程中的状态等。UIViewPropertyAnimator实例持有一个动画集，并且可以随时添加进新的动画。

> 注意: “UIViewPropertyAnimator实例”有点拗口，下面我会使用animator来表示它。

若有多个动画改变了一个view的同一个属性，则之后添加或开始*的动画“胜出”(ps:应该是有效的意思)。有趣的是不会发生不协调的变化，而是将新旧动画结合在了一起，当旧动画淡出时新动画淡入。

* 后来向UIViewPropertyAnimator实例中添加的动画或者是有一个开始延迟覆盖了之前的动画。"后开始者"胜出。

可中断，可逆的动画

来深入研究下，做个简单的动画。这个动画的效果是：当圆圈被拖拽时，会慢慢扩大到原始尺寸的两倍，一松手就会缩回原始大小。

首先，在类中添加animator属性与一个持续时间:

```
// ...
class ViewController: UIViewController {
    // We will attach various animations to this in response to drag events
    var circleAnimator: UIViewPropertyAnimator!
    let animationDuration = 4.0
// ...
```

需要在viewDidLoad中初始化animator:

```
// ...
// animations argument is optional
circleAnimator = UIViewPropertyAnimator(duration: animationDuration, curve: .easeInOut
, animations: {
[unowned circle] in
    // 2x scale
    circle.transform = CGAffineTransform(scaleX: 2.0, y: 2.0)
})
// self.view.addSubview(circle) here
// ...
```

当我们初始化circleAnimator时将duration与curve属性作为参数输入。curve可以设置为4个预置的时间变化曲线，在这个例子中选用easeInOut，其它的还有easeIn, easeOut和linear。还需要输入一个倍增圆圈尺寸的闭包函数。

现在需要一个触发动画的方法。使用下面的代码替换dragCircle方法的内容，当用户开始拖拽时开始动画，通过设定circleAnimator.isReversed的值来控制动画的方向。

```
func dragCircle(gesture: UIPanGestureRecognizer) {
    let target = gesture.view!

    switch gesture.state {
    case .began, .ended:
        circleCenter = target.center

        if (circleAnimator.isRunning) {
            circleAnimator.pauseAnimation()
            circleAnimator.isReversed = gesture.state == .ended
        }
        circleAnimator.startAnimation()

        // Three important properties on an animator:
        print("Animator isRunning, isReversed, state: \(circleAnimator.isRunning), \(c
ircleAnimator.isReversed), \(circleAnimator.state)")
    case .changed:
        let translation = gesture.translation(in: self.view)
        target.center = CGPoint(x: circleCenter!.x + translation.x, y: circleCenter!.y
 + translation.y)
    default: break
    }
}
```

执行动画，让"圆圈"呼吸起来，按下一段时间试试..

一个关键点

在动画执行后:

没有变化，而是卡在了扩大后的尺寸。

那么到底发生了什么? 简而言之就是当动画完成时丢弃了引用动画。

Animators有三种状态:

- inactive: 初始状态。当动画达到终点时animator会返回这个状态(可转换为active)。
- active: 动画运行时的状态(可转换为stopped或inactive)。
- stopped: 当调用stopAnimation:方法时animator的状态(返回到inactive)。

如图所示:

(来源 UIViewAnimating protocol reference)

任何指向inactive状态的转换会使所有animator中的动画被清除。

尝试了startAnimation方法，接下来研究一下其它两个。

为了让圆圈动起来，需要改变circleAnimator的初始化代码:

```
expansionAnimator = UIViewPropertyAnimator(duration: expansionDuration, curve: .easeIn
Out)
```

…修改dragCircle:

```
// ...
// dragCircle:
case .began, .ended:
    circleCenter = target.center

    if circleAnimator.state == .active {
        // reset animator to inactive state
        circleAnimator.stopAnimation(true)
    }

    if (gesture.state == .began) {
        circleAnimator.addAnimations({
            target.transform = CGAffineTransform(scaleX: 2.0, y: 2.0)
        })
    } else {
        circleAnimator.addAnimations({
            target.transform = CGAffineTransform.identity
        })
    }

case .changed:
// ...
```

现在，当用户开始或停止拖拽时，停止或终止animator(如果它是active状态的话)，animator会清除所有关联动画返回到inactive状态。这样我们就可以关联一个新的动画向圆圈发送期望的最终状态。

使用变换来改变视图样式的一个好处就是你可以通过CGAffineTransform.identity来轻松地重置视图的样式，不需要保留旧值。

注意，circleAnimator.stopAnimation(true)与下面的代码是等效的:

```
circleAnimator.stopAnimation(false) // 未完成 (停留在stopped状态)
circleAnimator.finishAnimation(at: .current) // 设置视图的实际属性为这一刻的变换值
```

finishAnimationAt:方法接受一个UIViewAnimatingPosition值。若输入start或end，则圆圈最终会变换为起始动画或终止动画。

关于持续时间

这里有一个微妙的小bug，每次我们停止一个动画并开始一个新的时，不论视图离最终变换的目标还差多少，新的动画都会持续4秒。

可以这样来修复:

```
// dragCircle:
// ...
case .began, .ended:
    circleCenter = target.center

    let durationFactor = circleAnimator.fractionComplete // Multiplier for original du
ration
    // multiplier for original duration that will be used for new duration
    circleAnimator.stopAnimation(false)
    circleAnimator.finishAnimation(at: .current)

    if (gesture.state == .began) {
        circleAnimator.addAnimations({
            target.backgroundColor = UIColor.green()
            target.transform = CGAffineTransform(scaleX: 2.0, y: 2.0)
        })
    } else {
        circleAnimator.addAnimations({
            target.backgroundColor = UIColor.green()
            target.transform = CGAffineTransform.identity
        })
    }

    circleAnimator.startAnimation()
    circleAnimator.pauseAnimation()
    // set duration factor to change remaining time
    circleAnimator.continueAnimation(withTimingParameters: nil, durationFactor: durati
onFactor)
case .changed:
// ...
```

现在，我们显式的停止了animator，根据方向来关联2个中的一个动画，并重启animator，使用continueAnimationWithTimingParameters:durationFactor:调整剩余的持续时间。这样的话就不会以完整的时间执行原始动画，而是一个`瘦身`过的短时间的扩展动画。continueAnimationWithTimingParameters:durationFactor:方法也被用来修改animator的timing函数*。

* 当输入一个新的timing函数时，旧timing函数中的变换会被插入进来。若将一个弹性timing函数变成一个线性的话，举个例子，动画在变得平滑之前会保持 `弹性` 一段时间。

Timing函数

新的timing函数比之前的要好很多。

旧的 **UIViewAnimationCurve** 参数依旧可用(比如之前用过的easeInOut)，这里有两个新的可用timing对象：**UISpringTimingParameters** 与 **UICubicTimingParameters**

UISpringTimingParameters

要使用UISpringTimingParameters实例需要设置衰减率(damping ratio)、质量(mass)、刚度(stiffness)与初始速率(initial velocity)，回使用这些参数通过一些公式计算来得到一个逼真的弹性动画。animator在初始化时可以同时接受UISpringTimingParameters和duration参数，不过后者会被忽略掉，不会参与公式计算，这解决了一些以前的sping动画函数中的弊病。

来做些不同的事情，使用sping animator把圆圈吸附在屏幕的中央。

ViewController.swift

```
import UIKit

class ViewController: UIViewController {
    // this records our circle's center for use as an offset while dragging
    var circleCenter: CGPoint!
    // We will attach various animations to this in response to drag events
    var circleAnimator: UIViewPropertyAnimator?

    override func viewDidLoad() {
        super.viewDidLoad()

        // Add a draggable view
        let circle = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 100.0, height: 100.0)
)
        circle.center = self.view.center
        circle.layer.cornerRadius = 50.0
        circle.backgroundColor = UIColor.green()

        // add pan gesture recognizer to circle
        circle.addGestureRecognizer(UIPanGestureRecognizer(target: self, action: #sele
ctor(self.dragCircle)))

        self.view.addSubview(circle)
    }

    func dragCircle(gesture: UIPanGestureRecognizer) {
```

```
        let target = gesture.view!

        switch gesture.state {
        case .began:
             if circleAnimator != nil && circleAnimator!.isRunning {
                circleAnimator!.stopAnimation(false)
            }
            circleCenter = target.center
        case .changed:
            let translation = gesture.translation(in: self.view)
            target.center = CGPoint(x: circleCenter!.x + translation.x, y: circleCente
r!.y + translation.y)
        case .ended:
            let v = gesture.velocity(in: target)
            // 500 is an arbitrary value that looked pretty good, you may want to base
 this on device resolution or view size.
            // The y component of the velocity is usually ignored, but is used when an
imating the center of a view
            let velocity = CGVector(dx: v.x / 500, dy: v.y / 500)
            let springParameters = UISpringTimingParameters(mass: 2.5, stiffness: 70,
damping: 55, initialVelocity: velocity)
            circleAnimator = UIViewPropertyAnimator(duration: 0.0, timingParameters: s
pringParameters)

            circleAnimator!.addAnimations({
                target.center = self.view.center
            })
            circleAnimator!.startAnimation()
        default: break
        }
    }
}
```

拖拽圆圈然后松开，它不仅会弹回初始点的位置，并且依旧会保持松手时的动量。这是由于在初始化sping timing的参数时给initialVelocity设置了一个速率:

```
// dragCircle: .ended:
// ...
let velocity = CGVector(dx: v.x / 500, dy: v.y / 500)
let springParameters = UISpringTimingParameters(mass: 2.5, stiffness: 70, damping: 55,
 initialVelocity: velocity)
circleAnimator = UIViewPropertyAnimator(duration: 0.0, timingParameters: springParamet
ers)
// ...
```

- 为了体现动画执行的路径，每隔一段时间就在主圆圈的中心点绘制一个小圆圈。可以看出，“直线”路径有些弯曲，因为在松开圆圈时会保留一些动量，并且由于弹性被拉向屏幕中心。

UICubicTimingParameters

UICubicTimingParameters允许你添加自定义三次贝塞尔曲线的控制点，需要注意的是超过0.0 - 1.0 范围的坐标点会被调整到这个范围内:

```
// Same as setting the y arguments to 0.0 and 1.0 respectively
let curveProvider = UICubicTimingParameters(controlPoint1: CGPoint(x: 0.2, y: -0.48),
controlPoint2: CGPoint(x: 0.79, y: 1.41))
expansionAnimator = UIViewPropertyAnimator(duration: expansionDuration, timingParamete
rs: curveProvider)
```

若你不喜欢这些提供的timing曲线，可以在满足UITimingCurveProvider协议的基础上来实现自己想要的时序曲线。

擦除动画

可以手动设置一个动画暂停的进度，通过使用animator的fractionComplete属性，输入一个0.0 - 1.0* 范围内的值。比如说值为0.5的话，则会将动画属性折半赋给最终的值，并且会忽略timing曲线。注意你设置的位置会被映射到当你重启动画时的timing曲线，因此值为0.5的fractionComplete属性并不意味着剩余的持续时间等于原始持续时间。

来做个别的例子。首先，在viewDidLoad:顶部初始化animator，然后添加两个动画:

```
// viewDidLoad:
// ...
circleAnimator = UIViewPropertyAnimator(duration: 1.0, curve: .linear, animations: {
    circle.transform = CGAffineTransform(scaleX: 3.0, y: 3.0)
})

circleAnimator?.addAnimations({
    circle.backgroundColor = UIColor.blue()
}, delayFactor: 0.75)
// ...
```

这次没有使用startAnimation，当动画进行时圆圈会逐渐变大并且在进行到75%的进度时变为蓝色。

同样需要实现一个新的dragCircle:方法:

```
func dragCircle(gesture: UIPanGestureRecognizer) {
    let target = gesture.view!

    switch gesture.state {
    case .began:
        circleCenter = target.center
    case .changed:
        let translation = gesture.translation(in: self.view)
        target.center = CGPoint(x: circleCenter!.x + translation.x, y: circleCenter!.y
 + translation.y)

        circleAnimator?.fractionComplete = target.center.y / self.view.frame.height
    default: break
    }
}
```

现在当拖拽圆圈时会根据它垂直方向的坐标来更新animator的fractionCompleteNow属性:

这里我用的是线性的timing曲线，不过在这个例子中用其它或者提供的timing曲线的话效果会更好。圆圈颜色变蓝的动画伴随一个压缩版的animatortiming曲线。

* 若自定义的animator支持越过端点的动画，则可以接受超出0.0 - 1.0范围外的值。

Pattern Matching in Swift

Swift中的模式匹配

> - 原文链接：Pattern Matching in Swift
> - 原文作者：Cosmin Pupaza
> - 译者：yrq110

不论对于哪个编程语言而言，模式匹配都是一个很有用的特性，根据这个特性使你可以设计规则来匹配期望的值，使代码更加灵活与简洁。

苹果在Swift中加入了模式匹配，今天你将会看到Swift中的模式匹配技术。

教程中涵盖的模式:

- 元组模式
- 类型转换模式
- 通配符模式
- 可选模式
- 枚举用例模式
- 表达式模式

为了体现模式匹配的有用之处，你在这篇教程中会以一个独特的视角: 作为raywenderlich.com的主编！使用模式匹配来安排与发布网站上的教程。

> 注意: 这篇教程需要Xcode 8 与Swift 3，并且假定你已经掌握了基本的Swift开发。如果你是Swift新手请先看看我们的其它Swift教程。

入门

欢迎，临时主编! 你今天的主要任务是安排在网站上发布的教程。下载开始的plauground并在Xcode中打开**starter project.playground**。

playground中包含两个东西:

- **random_uniform(value:)**函数，会从0与一个实际值间返回一个随机数，使用它来生成

随机日期。
- 解析**tutorials.json**文件将内容转换为字典数组并返回的样板代码。使用它来提取需要安排的教程信息。

> 注意：了解更多有关Swift的JSON解析可以查看我们的教程。

无需理解它们是如何工作的，只需要知道文件结构即可，接下来打开playground资源文件夹中的tutorials.json文件。

每一篇安排的教程都有两个属性: 标题与日期。你的老大为你安排好了发布的日程，每天都分配了一篇教程，1表示周一，5表示周五，依次类推，nil表示不在日程内。

你想将本周的课程安排为每天发布一篇教程，然而在查看日程表时发现你的老大把两篇教程安排到了同一天。需要你来修正这个问题，你想通过一个特定的规则来对这些教程进行排序，该怎么办？

如果你想到了“使用模式!” 那么就上道儿了。 :]

模式匹配类型

了解一下这篇教程中需要用到的模式类型。

- 元组模式 用于匹配元组类型的值。
- 类型转换模式 允许你转换或匹配类型。
- 通配符模式 用于匹配并忽略任何类型的值。
- 可选模式 用于匹配可选值。
- 枚举用例模式 匹配存在的枚举类型的用例。
- 表达式模式 允许你使用表达式与值进行比较。

在成为一个前所未有的优秀主编的道路上你会使用所有这些模式！

让这家伙失业吧!(PS:此人为来源站主编)

元组模式

首先要出创建一个元组模式来生成一个教程的数组。在playground的底部添加如下代码:

```
enum Day: Int {
  case monday, tuesday, wednesday, thursday, friday, saturday, sunday
}
```

创建了一个包含一周七天的枚举，由于初始值是Int类型的，因此每一天都被分配了一个Int型值，Monday是0，Sunday是6，以此类推。

在枚举声明的后面添加如下代码:

```
class Tutorial {

  let title: String
  var day: Day?

  init(title: String, day: Day? = nil) {
    self.title = title
    self.day = day
  }
}
```

这里定义了一个包含两个属性的tutorial类型:tutorial的标题与发布日期。日期是一个可选变量，因此应未发布的教程来说它可以是nil。

实现CustomStringConvertible，这样就可以轻松的将教程信息打印出来了:

```
extension Tutorial: CustomStringConvertible {
  var description: String {
    var scheduled = ", not scheduled"
    if let day = day {
      scheduled = ", scheduled on \(day)"
    }
    return title + scheduled
  }
}
```

添加一个数组来存放教程:

```
var tutorials: [Tutorial] = []
```

接着，通过在playground底部添加如下代码，将开始工程中的字典数组转换为教程数组:

```
for dictionary in json {
  var currentTitle = ""
  var currentDay: Day? = nil

  for (key, value) in dictionary {
    // todo: extract the information from the dictionary
  }

  let currentTutorial = Tutorial(title: currentTitle, day: currentDay)
  tutorials.append(currentTutorial)
}
```

在这里使用for-in语句遍历了json数组。使用以元组格式为循环值的for-in语句遍历了数组中每个字典的键值对。这就是元组模式的实现。

你把每个教程都添加进了数组中，不过当前教程的属性是空的，需要在下一章节中使用类型转换模式来设置教程的属性。

类型转换模式

为了从字典中提取教程信息，需要一个类型转换模式。在for-in循环中添加如下代码来替换掉注释:

```
// 1
switch (key, value) {
  // 2
  case ("title", is String):
    currentTitle = value as! String
  // 3
  case ("day", let dayString as String):
    if let dayInt = Int(dayString), let day = Day(rawValue: dayInt - 1) {
      currentDay = day
  }
  // 4
  default:
    break
}
```

逐步分析下:

1. 切换到键值元组---重载的元组模式。
2. 使用 `is` 类型转换模式检测教程的标题是否属于String类型，若是则转换它。
3. 使用 `as` 类型转换模式检测教程的日期是否是String类型，若是则先将其转换成Int，然后传入枚举的可失败构造器init(rawValue:)中，得到对应的枚举值。将dayInt减去1是因为枚举的初始值是0，而tutorial.json中是从1开始的。

4. switch语句应该是完备的，需要添加一个default case，使用break语句来跳出switch。

在playground的底部添加如下代码，在控制台打印数组中的内容:

```
print(tutorials)
```

如你所见，现在数组中的每个教程都含有名称与发布日期了。万事俱备只欠完成的任务: 每周每天仅发布一篇教程。

通配符模式

使用通配符来安排教程的发布，不过需要先将它们的发布日期置空。在playground底部添加如下代码:

```
tutorials.forEach { $0.day = nil }
```

将数组中的所有教程的日期设为nil，为了安排教程的日期，在playground的底部添加如下代码段:

```
// 1
let days = (0...6).map { Day(rawValue: $0)! }
// 2
let randomDays = days.sorted { _ in random_uniform(value: 2) == 0 }
// 3
(0...6).forEach { tutorials[$0].day = randomDays[$0] }
```

这儿有点复杂，分解成如下步骤看下:

1. 首先创建一个有关天数的数组，包含每周中不重复的每一天。
2. “整理”这个数组， random_uniform(value:)函数用来生成随机数。在闭包中，使用下划线来忽略闭包参数，因为并不需要它。虽然有在技术上更加高效、在数学上更加准确的方法来随机变换数组元素位置，不过这就是一个通配符模式的实现!
3. 最后，将一周中七天随机分配给了最开始的七个教程。

在playground底部添加下列代码，控制台中打印安排好的教程:

```
print(tutorials)
```

成功了! 现在为一周中每一天都安排了一篇教程，日程中既没有发布时间重合的教程也没有时间空隙。干得好!

可选模式

日程表被占用了，作为主编你需要整理一下这些教程。The schedule has been conquered, but as editor-in-chief you also need to sort the tutorials. 使用可选模式来处理它。You'll tackle this with optional patterns. 将教程数组按升序排序，首先根据标题来整理未安排的教程，其次根据日期来整理来已安排的教程，在playground底部添加如下代码:

```
// 1
tutorials.sort {
  // 2
  switch ($0.day, $1.day) {
    // 3
    case (nil, nil):
      return $0.title.compare($1.title, options: .caseInsensitive) == .orderedAscendin
g
    // 4
    case (let firstDay?, let secondDay?):
      return firstDay.rawValue < secondDay.rawValue
    // 5
    case (nil, let secondDay?):
      return true
    case (let firstDay?, nil):
      return false
  }
}
```

逐步分析下:

1. 使用sort(_:)方法来整理教程数组，方法的参数是一个结尾闭包(trailing closure)，在闭包中定义了数组中两个教程的排序顺序。若将教程按升序进行排序则返回true，否则返回false。
2. 切换到由当前被整理的两个教程组成的元组中，这又是一个 `元组模式` 。
3. 若两个教程都未被安排，它们的day属性是nil，则使用数组的 `compare(\_:options:)` 方法按标题的升序来整理它们。
4. 使用 `可选模式` 来检测两个教程是否都被安排了，这种模式仅匹配可以被解绑的值，若两个值都可被解绑，则按它们的raw value的升序来整理。
5. 再次使用可选模式，检测只有一个教程被安排时的情况，若满足，则优先整理未被安排的教程。 在playground的最后添加这行代码，打印出整理后的教程:

```
print(tutorials)
```

现在已经将教程按你想要的方式排序好了，做的这么好理应涨工资! 然而 ... 还需要做的事有很多。

枚举用例模式

现在让我们来使用枚举用例模式来决定每个教程发布天的名称。

在Tutorial的扩展中，使用了Day类型中的枚举用例名称来构建自定义字符串。添加一个合适的名称，而不是与原有名称绑定，在playground的底部添加如下代码:

```
extension Day {

  var name: String {
    switch self {
      case .monday:
        return "Monday"
      case .tuesday:
        return "Tuesday"
      case .wednesday:
        return "Wednesday"
      case .thursday:
        return "Thursday"
      case .friday:
        return "Friday"
      case .saturday:
        return "Saturday"
      case .sunday:
        return "Sunday"
    }
  }
}
```

switch语句中使用当前值(self)来匹配枚举用例，这就是枚举用例模式的实现。

印象挺深的，不是吗? 虽然用数字挺酷的，不过毕竟用名称的话很直观并且更容易理解! :]

表达式模式

接着来添加一个描述教程发布顺序的属性。也许你会再次使用枚举用例模式，如下(别在**playground**中添加**!**):

```
var order: String {
  switch self {
    case .monday:
      return "first"
    case .tuesday:
      return "second"
    case .wednesday:
      return "third"
    case .thursday:
      return "fourth"
    case .friday:
      return "fifth"
    case .saturday:
      return "sixth"
    case .sunday:
      return "seventh"
  }
}
```

不过同一件事做两遍的话就太不像主编了，不是吗? ;] 另辟蹊径，使用表示式模式。首先需要重载模式匹配的操作符，这是为了改变默认的功能并且使其对Day属性有效。在playground底部添加如下代码:

```
func ~=(lhs: Int, rhs: Day) -> Bool {
  return lhs == rhs.rawValue + 1
}
```

这段代码用来匹配day与整数，会返回数字1~7。使用这个重载的操作符以不同的方法来得到计算后的属性。

在playground底部添加如下代码:

```
extension Tutorial {

  var order: String {
    guard let day = day else {
      return "not scheduled"
    }
    switch day {
      case 1:
        return "first"
      case 2:
        return "second"
      case 3:
        return "third"
      case 4:
        return "fourth"
      case 5:
        return "fifth"
      case 6:
        return "sixth"
      case 7:
        return "seventh"
      default:
        fatalError("invalid day value")
    }
  }
}
```

多亏了重载的模式匹配操作符，day对象可以使用整数表达式来匹配了。这就是表达式模式的实现。

整合

现在已经定义了day的名称与教程的顺序，可以打印出每个教程的信息看看。在playground底部添加如下代码:

```
for (index, tutorial) in tutorials.enumerated() {
  guard let day = tutorial.day else {
    print("\(index + 1). \(tutorial.title) is not scheduled this week.")
    continue
  }
  print("\(index + 1). \(tutorial.title) is scheduled on \(day.name). It's the \(tutor
ial.order) tutorial of the week.")
}
```

注意for-in语句中的元组了吗? 这又是一个元组模式!

喔! 对于你的一日主编来说真是够忙啊，不过你做的不错，现在可以在泳池里放松休息一下了。

开个玩笑!主编的活儿是忙不完的。回去工作!

最后

最终的playground下载。

CloudKit Tutorial: Getting Started

CloudKit指南：入门

> - 原文链接：CloudKit Tutorial: Getting Started
> - 原文作者：Ed Sasena
> - 译者：yrq110

> 更新: 此教程已更新至 Xcode 8, iOS 10, Swift 3。原教程由Michael Katz所著。

CloudKit是苹果基于iCloud的远程数据存储服务。提供了一个低开销的存储与分享app数据的方案，使用用户的iCloud账户进行后端存储。

学习如何使用CloudKit在云端存储你的app数据!

CloudKit中的两个主要部分:

1. 一个管理record type与其它公共数据的web控制台。
2. 一个在iCloud与设备之间传输数据的API集合。

CloudKit同样很安全，用户的私有数据被完全保护了起来，开发者只能访问他们自己的私有数据库而不能查看用户的私有数据。

对于一个仅有iOS平台并且使用大量数据的app来说，CloudKit是一个好的选择，无需编写服务端的一些繁琐的逻辑。此外，CloudKit也可以应用于web与服务器应用。

在这篇CloudKit教程中，会使用CloudKit创建一个餐厅评分app来进行实践，它有一个拗口的名字-BabiFüd.

> 注意: 进行这篇CloudKit教程中的示例app开发时，需要一个激活的iOS开发者账号，若没有的话将无法使用iCloud entitlements与访问CloudKit控制台。

为何使用CloudKit?

也许你会疑惑为何不选Core Data或其他商业型BaaS服务，甚至是自己的服务器。

这里有三个原因: 简便, 信任, 与开销。

简便

不像其它的后端解决方案，CloudKit仅需要少量的配置即可使用。无需进行服务器的选择、配置与安装，并且安全问题也是由苹果来处理。

进行简单的iOS开发者注册后即可使用CloudKit，不需要注册额外的服务或创建新账号，当在app中启用CloudKit功能时所有必要的服务器都会神奇的自动设置完成。

无需下载额外的库与配置。CloudKit与其它iOS框架同等重要，CloudKit框架本身也提供了一些针对于通用操作的方便的API。

对用户来说很简单，CloudKit在设备启动好时通过iCloud证书来登入(或者在启动后的设置界面登入)，无需构建复杂的登录界面。只要登入成功后，就可以直接使用你的app了。That should put you on Cloud 9(PS: 也许指的是北美的那支游戏战队)!

信任

另一个使用CloudKit的好处就是用户可以放心它们数据的隐私性与安全性，这些数据都是由苹果处理的而不是app开发者。CloudKit将用户数据与开发者隔离开来。

虽然这个访问权限的缺失在debug过程中也许会使人沮丧，不过你不用担心安全性并且思考如何使用户相信他们的数据是安全的。若用户信任iCloud的话他们也会信任你。

开销

最后，对开发者而言，运行一个服务总会产生一笔不小的开销，即使是价格最便宜的服务器主机也很难有为小型免费的app提供的低开销解决方案。因此在运行一个app时难免会产生一个相关的开销。

使用CloudKit的话会得到一个免费的用于公共数据传输的容量适当的存储空间。在WWDC 2015视频中的What’s New in CloudKit详细讲解了它的费用。

强有力的CloudKit能为Mac与iOS app提供一个麻烦少的解决方案。

BabiFüd简介

BabiFüd是这篇教程中的示例app，一个标准的“餐厅评价”类app。然而不通过食物质量、服务速度与价格来进行评估，而是通过针对儿童的友好度，这就包含了可更换的设施，安全座椅与可选择的健康食品选择。

app中包含4个标签：一个附近餐厅的列表，一张附近餐厅的地图，用户评价与设置。附近餐厅的列表是唯一会在教程中用到的，看看下面的app截图

这些视图后有一个模型类，与调用CloudKit的操作绑定。CloudKit对象被称为records，模型中主要的record类型是**Establishment**，代表app中不同的餐厅。

开始

先下载这篇教程的开始工程。

在开始编码前需要修改下Bundle Identifier和app的Team属性，需要设置team来从Apple那里获权，给予一个独一无二的bundle identifier会省不少事儿。

在Xcode中打开BabiFud.xcodeproj，工程导航栏中选择BabiFud工程，接着选择BabiFud target，选中General标签，修改Bundle Identifier，一个经典的做法是反转标记的区域名称再加上工程名称，接着选择适合的Team:

好了，现在需要设置CloudKit，创建一些存放数据的容器(container)。

授权与容器

通过app添加数据前需要一个存放app记录(record)的容器。容器是存放在服务器的保存所有app数据的一个概念性位置，分为公有数据库与私有数据库。

为了创建一个容器，首先需要启用app的iCloud功能，选择target中的Capabilities标签，将iCloud区域的开关拨到ON。

这时，Xcode可能会让你输入与iOS开发者账号关联的Apple ID，若有的话就输入下。最后，在Services分组里勾选CloudKit复选框来启用CloudKit。

这样就创建了一个默认的名为iCloud.<你的bundle id>的容器，如下所示:

Xcode中配置iCloud的一些问题

如果你在创建entitlement，构建构成，运行app或者Xcode提示容器ID时遇到警告或错误时，这里有一些提示:

- 若在iCloud部分设置时出现警告或错误，试试点击Fix Issue按钮，可能需要多试几次。
- 有一点很重要：app的bundle id需要与iCloud容器相对应，并且在开发者账号中是存在

的。比如说，bundle id是“com..BabiFud”，则iCloud容器名称就应该是“iCloud.”加上bundle id变成“iCloud.com..BabiFud”。

- iCloud容器名必须是唯一的，因为这是Cloudkit用来访问数据所使用的全局标识符。由于iCloud容器名包含bundle id，因此bundle id也必须是唯一的(这就是为何需要修改com.raywendrelich.BabiFud)。
- 为了让entitlements起作用，需要在App的证书、标识符与配置文件中ID的部分列出app/bundle id。这意味着标识的证书使用了设置的team id与app id，从中可得到iCloud容器的id。若已经在一个可用的开发者账号中标识了的话Xcode会自动完成这一切。不巧的是，这有时是不同步的，需要更新ID-使用iCloud功能面板修改CloudKit容器ID。否则的话需要修改info.plist文件或BabiFud.entitlements文件来确保id values与所设置的bundle id一致。

CloudKit控制台介绍

设置完必要的entitlements后，下一步就是创建与定义app所使用的record type，可以使用CloudKit控制台来做这件事。在target的Capabilities面板中，在iCloud下方点击CloudKit Dashboard即可。

> 提示: 也可以在浏览器中用URL打开CloudKit控制台
> https://icloud.developer.apple.com/dashboard/

控制台长这样:

CloudKit控制台由四个部分组成: Schema(方案), Public Data(公有数据), Private Data(私有数据), and Admin(用户)。

SCHEMA区域展示了CloudKit容器的高级对象: Record Types(记录类型), Security Roles(安全规则), and Subscription Types(订阅类型). 在此教程中只需要关注Record Type。You'll only be concerned with Record Types in this tutorial.

一个Record Type就是定义了一些record个体的集合。就像面向对象编程中的class。一个record可以作为一个特殊的Record Type实例，表示容器中的结构化数据，就像数据库中的一行数据，封装了一系列的键值对。

PUBLIC DATA与PRIVATE DATA区域允许你添加、搜索数据。记住，作为一个开发者可以访问所有公有数据与自己的私有数据。User Records存储当前iCloud用户数据，比如姓名与邮箱。 Record Zone中通过进行分组记录来给私有数据库提供一个逻辑性的组织结构。Custom zones支持原子事务(PS:原子事务：web服务上的操作或者全部发生，或者根本不发生。)，允许多个记录在进行其它操作前同时存储。此教程中不涉及Custom zones。

ADMIN区域允许你配置team成员的控制台访问权限。若你的team中有多个开发者同伴，可以在这里编辑他们的权限。同样，这个也不是此教程的内容。

添加商家(Establishment)的Record Type

考虑一下app中如何设计，每一条想要获取的信息都含有多个数据：名称，地点，还有多种针对儿童友好度的选项。Record types使用字段来定义记录中的多种数据。

选择Record Types，点击左上角的 + 号添加一个新的record type。

将这个新的record type命名为Establishment。

会看到一行字段，可以在这里定义Field Name(字段名), Field Type(字段类型) 与 Index(索引)，如下所示。 自动创建的的默认字段名为StringField。

首先替换字段名，字段类型与索引在首次定义时是默认的值，需要改变一些字段。点击**Add Field…** 添加所需的字段，如下所示:

搞定后列表应像下面这样:

点击页面上方的Save进行保存。

好了，现在在数据库中添加一些样本记录。

选择PUBLIC DATA区域的Default Zone，这个区域包含app中的公有记录。从中间面板的列表中选择Establishment记录类型，选择 + 图标或右侧细节面板的New Record按钮，如下图所示:

会创建一个新的Establishment记录。

这时需要录入一些用于app测试的数据。

下面的这些商家数据是虚构的，都设定在Apple总部的附近，容易在模拟器中找到。

录入下面这些记录:

> 注意: 图中的CoverPhoto元素在Xcode项目中的Supporting Files\Sample Images文件夹。将其拖拽到CoverPhoto字段即可添加到establishment record中。

保存完成后，在控制台中应如下所示:

对于每条记录，输入的值都是数据库中所表现的数据。在app端数据类型是不同的。比如说SeatingType与ChangingTable是结构体，给SeatingType分配的整型值对应“high chair”或“booster”。对于HealthyOption与KidsMenu, 用整型值来表示布尔值，0意味着商家没有这个选项，1则是有。

运行app时需要你有一个用于开发的iCloud账号。

创建一个开发用的iCloud账号

也需要在iOS模拟器中录入与这个账号关联的iCloud证书。

运行app前录入iCloud证书

回到Xcode中，是时候在app中整合数据了!

查询商家记录

使用CKQuery对象从数据库中选择记录，找到所有符合一定规则的记录，这个规则可以是“所有name是M开头的记录”，“包含增高座椅的记录”，“3km之内的记录”。这些表达式在Cocoa中使用NSPredicate对象来编写，NSPredicate通过匹配特殊的规则来评估对象是否符合要求。谓词也同样用在CoreData中，自然支持CloudKit。

CloudKit仅支持一部分可用的NSPredicate函数，包含数学的比较、一些字符串与设置的操作(例如“匹配列表中的一项的字段”)和一个特殊的距离函数。NSPredicate的distanceToLocation:fromLocation:函数通过与已知位置相距一个特殊半径的位置字段来匹配CloudKit中的记录，这种类型的谓词会在之后来详细说明。对于其他查询类型，看看苹果的官方文档CKQuery Class Reference，列表中包含了所支持的函数与它们的使用方法。

> 注意: CloudKit支持CLLocation对象(CoreLocation框架的对象，包含地理空间坐标)，这使得查询一个地理区域内的商家变得很简单 - 不需要做任何额外的坐标变换等操作。

在Xcode中打开Model/Model.swift，这个文件中包含了app中需要的所有调用服务器方法的存根。

使用如下代码替换fetchEstablishments(_:radiusInMeters:)方法:

```
func fetchEstablishments(_ location:CLLocation, radiusInMeters:CLLocationDistance) {
  // 1
  let radiusInKilometers = radiusInMeters / 1000.0
  // 2
  let locationPredicate = NSPredicate(format: "distanceToLocation:fromLocation:(%K,%@)
 < %f", "Location", location, radiusInKilometers)
  // 3
  let query = CKQuery(recordType: EstablishmentType, predicate: locationPredicate)
  // 4
  publicDB.perform(query, inZoneWith: nil) { [unowned self] results, error in
    if let error = error {
      DispatchQueue.main.async {
        self.delegate?.errorUpdating(error as NSError)
        print("Cloud Query Error - Fetch Establishments: \(error)")
      }
      return
    }
    self.items.removeAll(keepingCapacity: true)
    results?.forEach({ (record: CKRecord) in
      self.items.append(Establishment(record: record,
                                      database: self.publicDB))
    })
    DispatchQueue.main.async {
      self.delegate?.modelUpdated()
    }
  }
}
```

逐步分析下:

1. CloudKit中距离谓词的单位是千米，这行代码将radiusInMeters的单位转换为千米。
2. 谓词过滤出的商家是基于离当前位置一定的距离得到的。这个语句根据用户当前位置找到所有一定距离内的商家。
3. 使用一个谓词与一个record type来创建CKQuery对象，两者在执行查询时都会被用到。
4. 最后，performQuery(_:inZoneWithID:completionHandler:)将你的查询提交给iCloud，并等待查询结果。将inZoneWithID参数设为nil，即在你的默认区域中执行查询，这个默认区域就是公有数据库。若你想同时在私有与公有数据库中检索数据，需要分别调用每个数据库的查询功能。

你也许会疑问CKDatabase实例与publicDB到底来自哪里？看看Model.swift的文件顶部。

```
let container: CKContainer
let publicDB: CKDatabase
let privateDB: CKDatabase

init() {
  // 1
  container = CKContainer.defaultContainer()
  // 2
  publicDB = container.publicCloudDatabase
  // 3
  privateDB = container.privateCloudDatabase
}
```

在这里定义了数据库:

1. 这个默认容器就是在iCloud Capabilities面板中的那个。
2. 公有数据库中的数据由所有app的用户共享。
3. 私有数据库仅包含当前登入用户的数据。

这段代码会从公有数据库中检索到一些本地的商家数据，不过需要将这些数据与一个view controller结合才能在app中显式的看到。

准备必要的回调函数

可以用熟悉的代理模式来编写notification函数，在Model.swift的顶部有需要在view controller中实现的协议:

```
protocol ModelDelegate {
  func errorUpdating(error: NSError)
  func modelUpdated()
}
```

打开MasterViewController.swift，使用下面的代码替换modelUpdated()方法:

```
func modelUpdated() {
  refreshControl?.endRefreshing()
  tableView.reloadData()
}
```

当有新的可用数据时会调用这个方法，在tableView(_:cellForRowAtIndexPath:)方法中已经将所有列表项与CloudKit对象绑定好了，可以如释重负的去瞥一眼。

接下来，在MasterViewController.swift中使用如下代码替换errorUpdating(_:)方法:

```
func errorUpdating(_ error: NSError) {
  let alertController = UIAlertController(title: nil,
                                          message: error.localizedDescription,
                                          preferredStyle: .alert)

  alertController.addAction(UIAlertAction(title: "Dismiss", style: .default, handler:
nil))

  present(alertController, animated: true, completion: nil)
}
```

当查询过程产生错误时会调用这个方法。较差的网络条件或者特殊的CloudKit问题(比如缺失或错误的用户证书、没有返回结果的查询)都会引起错误。

在操作各种远程服务器时一个良好的错误处理是很必要的，在这里仅仅给用户显示错误信息。

有个很常见的问题：用户未登入iCloud或者没有为app打开iCloud drive。你能修改一下errorUpdating(_:)来处理这几种情况吗？ 提示: 两种错误均会返回一个值为1的CKErrorCode。

解决方法：在MasterViewController.swift文件中使用如下代码替换errorUpdating(_:)方法:

```
func errorUpdating(_ error: NSError) {
  let message: String
  if error.code == 1 {
    message = "Log into iCloud on your device and make sure the iCloud drive is turned
 on for this app."
  } else {
    message = error.localizedDescription
  }
  let alertController = UIAlertController(title: nil,
                                          message: message,
                                          preferredStyle: .alert)

  alertController.addAction(UIAlertAction(title: "Dismiss", style: .default, handler:
nil))

  present(alertController, animated: true, completion: nil)
}
```

构建并运行，应该会看到一个附近商家的列表。

调试时的问题

如果你使用的是iOS模拟器，并且列表没有被填充，确认一下你的地理位置设置是正确的(Xcode菜单:Debug\Simulate Location\San Francisco, CA, USA)。若需要修改在Xcode中设置的位置，在app中下拉列表刷新一下位置信息。

若你使用的iPhone或iPad，位置服务是可用的并且商家位置并不在你当前位置的附近的话，此时有两个选择:改变样例数据的坐标使其位置靠近你当前的位置或者使用模拟器运行app。不过这里还有第三个选项：你可以去苹果总部附近转转。

若数据没有正确显示或显示不完全，则需要使用CloudKit控制台检查一下样例数据。为了保证所有记录都是存在的，要将它们添加到默认域中并且有正确的值。若要重新输入数据，可以点击垃圾箱图标删除记录。

有时修复CloudKit的错误是很棘手的。CloudKit的错误消息并不包含大量的信息，为了搞清楚错误发生的原因需要结合当前所操作数据库的错误代码。使用数字型错误代码，查找对应的CKErrorCode枚举值。查看文档中的名称与描述有助于找到问题所在，看看下面的例子。

> 注意: 可以在CloudKit Framework Constants Reference中查看CloudKit返回的错误代码列表。

这里是一些常见的错误枚举值与相关描述:

- .BadContainer – 指定的容器未知或未授权。
- .NotAuthenticated – 当前用户没有认证，没有可用的用户记录。当用户没有登入iCloud时可能会出现这个错误。
- .UnknownItem – 指定的记录不存在。

当抓取到商家列表数据时，会注意到可以看到商家名称与提供的服务，不过没有显示的图片，被云挡住了？

这是因为在检索商家记录的同时也会自动检索图片数据，不过你需要执行一些必要的步骤去加载图片才能显示出来，这样就可以让云散去了! :]

操作二进制资源

资源就是与记录(record)相关的二进制数据，比如说一张图片。对你而言，app中的资源是MasterViewController列表中的商家图片。

在这一部分中会添加一些逻辑代码，用来加载之前检索并下载的商家记录中的资源。

打开Model/Establishment.swift文件，使用如下代码替换loadCoverPhoto(_:)方法:

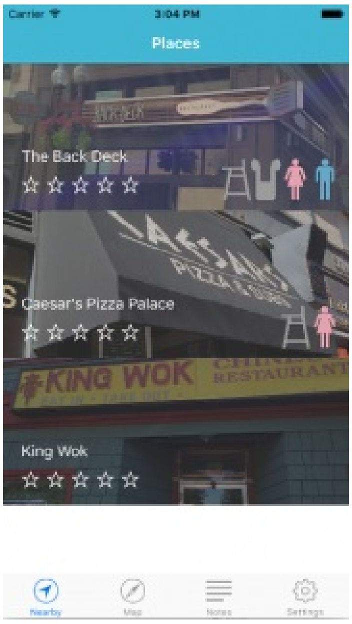
```
func loadCoverPhoto(completion:@escaping (_ photo: UIImage?) -> ()) {
  // 1
  DispatchQueue.global(qos: DispatchQoS.QoSClass.background).async {
    var image: UIImage!
    defer {
      completion(image)
    }
      // 2
    guard let asset = self.record["CoverPhoto"] as? CKAsset else {
      return
    }

    let imageData: Data
    do {
      imageData = try Data(contentsOf: asset.fileURL)
    } catch {
      return
    }
    image = UIImage(data: imageData)
  }
}
```

上述代码按下列步骤从资源属性中加载了图像数据:

1. 虽然在检索记录的同时下载了资源，还需要异步加载图片才行，因此在dispatch_async闭包中要搞定这一切。
2. 资源作为CKAsset实例存放在CKRecord中，适当的组合代码获取资源。接下来从本地文件URL提供的资源中加载图像数据。
3. 使用图像数据创建一个UIImage实例。
4. 使用检索到的图像数据执行completion回调函数。注意这个defer闭包在执行时是不管之前执行的是哪个return的（不会判断是否获得资源）。比如说没有图片资源时，会在没有设置image值的情况下return，那么就不会显示任何餐厅的图片。

构建并运行，这样商家图片就能正常显示了。干得好!



可以在这里下载最后的工程

Face Detection in iOS Using Core Image

使用Core Image的iOS人脸识别

> - 原文链接 : Face Detection in iOS Using Core Image
> - 原文作者 : GREGG MOJICA
> - 译者 : yrq110

Core Image是Cocoa Touch中一个强大的API，也是iOS SDK中的关键部分，然而它经常被人们忽视。在这篇教程中会讲解一些Core Image的人脸识别特性，并在你的iOS app中应用这项技术!

> 注意: 这是一篇中高级的iOS教程，假定你已经使用过UIImagePicker, Core Image等技术。若你不太熟悉这些技术，可以先看看我们的系列教程，准备好了再回来。

将要构建的应用

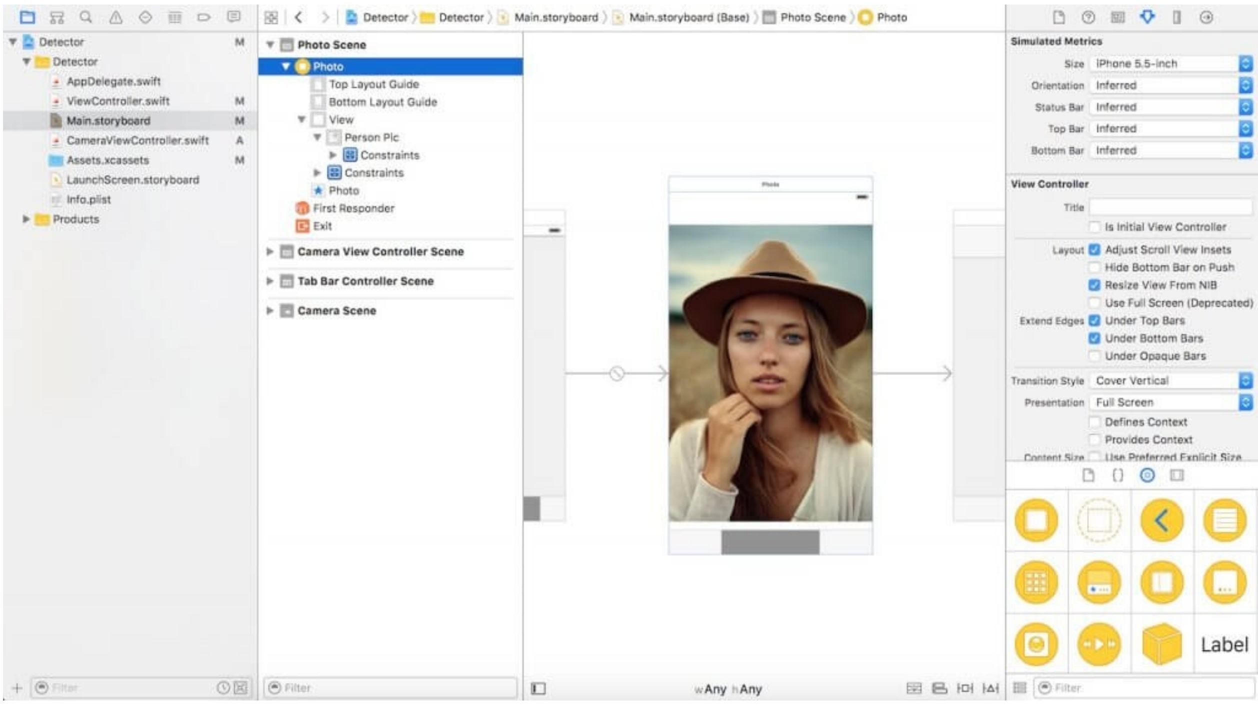
iOS的人脸识别从iOS 5(2011)就有了，不过一直没怎么被关注过。人脸识别API允许开发者不仅可以检测人脸，也可以检测到面部的一些特殊属性，比如说微笑或眨眼。

首先，为了了解Core Image的人脸识别技术我们会创建一个app来识别照片中的人脸并用一个方框来标记它。在第二个例子中，让用户拍摄一张照片，检测其中的人脸并检索人脸位置。这样一来，就充分掌握了iOS中的人脸识别，并且学会如何利用这个强大却总被忽略的API。

开搞!

建立工程

下载开始工程在Xcode中打开。如你所见，里面只有一个关联了IBOutlet和imageView的故事板。



> 感谢: 绑定的图像来自[unsplash.com](unsplash.com)。

使用Core Image识别人脸

在开始工程中，故事板中的imageView组件与代码中的IBOutlet已关联，接下来要编写实现人脸识别的代码部分。在swift文件中插入如下代码，之后进行讲解:

```
func detect() {

    guard let personciImage = CIImage(image: personPic.image!) else {
        return
    }

    let accuracy = [CIDetectorAccuracy: CIDetectorAccuracyHigh]
    let faceDetector = CIDetector(ofType: CIDetectorTypeFace, context: nil, options: a
ccuracy)
    let faces = faceDetector.featuresInImage(personciImage)

    for face in faces as! [CIFaceFeature] {

        print("Found bounds are \(face.bounds)")

        let faceBox = UIView(frame: face.bounds)

        faceBox.layer.borderWidth = 3
        faceBox.layer.borderColor = UIColor.redColor().CGColor
        faceBox.backgroundColor = UIColor.clearColor()
        personPic.addSubview(faceBox)

        if face.hasLeftEyePosition {
            print("Left eye bounds are \(face.leftEyePosition)")
        }

        if face.hasRightEyePosition {
            print("Right eye bounds are \(face.rightEyePosition)")
        }
    }
}
```

来探讨下发生了什么:

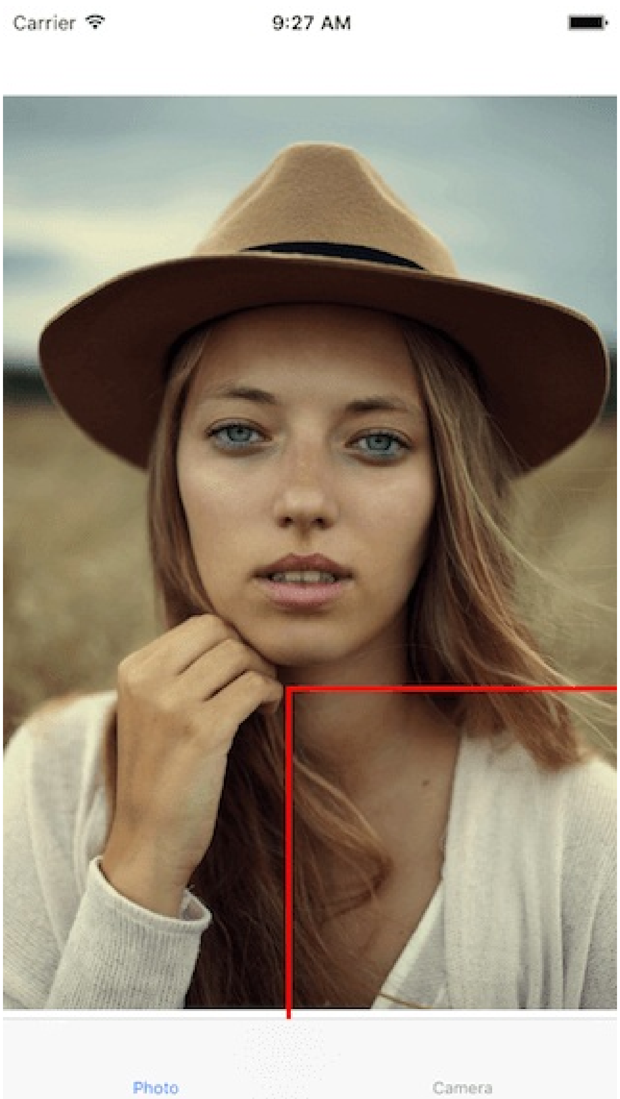
- 行 #3: 创建personciImage变量保存从故事板中的UIImageView提取图像并将其转换为CIImage，使用Core Image时需要用CIImage。
- 行 #7: 创建accuracy变量并设为CIDetectorAccuracyHigh，可以在CIDetectorAccuracyHigh(较强的处理能力)与CIDetectorAccuracyLow(较弱的处理能力)中选择，因为想让准确度高一些在这里选择CIDetectorAccuracyHigh。
- 行 #8: 这里定义了一个属于CIDetector类的faceDetector变量，并输入之前创建的accuracy变量。
- 行 #9: 调用faceDetector的featuresInImage方法，识别器会找到所给图像中的人脸，最后返回一个人脸数组。
- 行 #11: 循环faces数组里的所有face，并将识别到的人脸强转为CIFaceFeature类型。
- 行 #15: 创建名为faceBox的UIView，frame设为返回的faces.first的frame，绘制一个矩形框来标识识别到的人脸。
- 行 #17: 设置faceBox的边框宽度为3。

- 行 #18: 设置边框颜色为红色。
- 行 #19: 将背景色设为clear，意味着这个视图没有可见的背景。
- 行 #20: 最后，把这个视图添加到personPic imageView上。
- 行 #22-28: API不仅可以帮助你识别人脸，也可识别脸上的左右眼，我们不在图像中标识出眼睛，只是给你展示一下CIFaceFeature的相关属性。

需要在viewDidLoad中调用detect方法，插入如下行:

```
detect()
```

编译并运行app，结果应如下图所示:

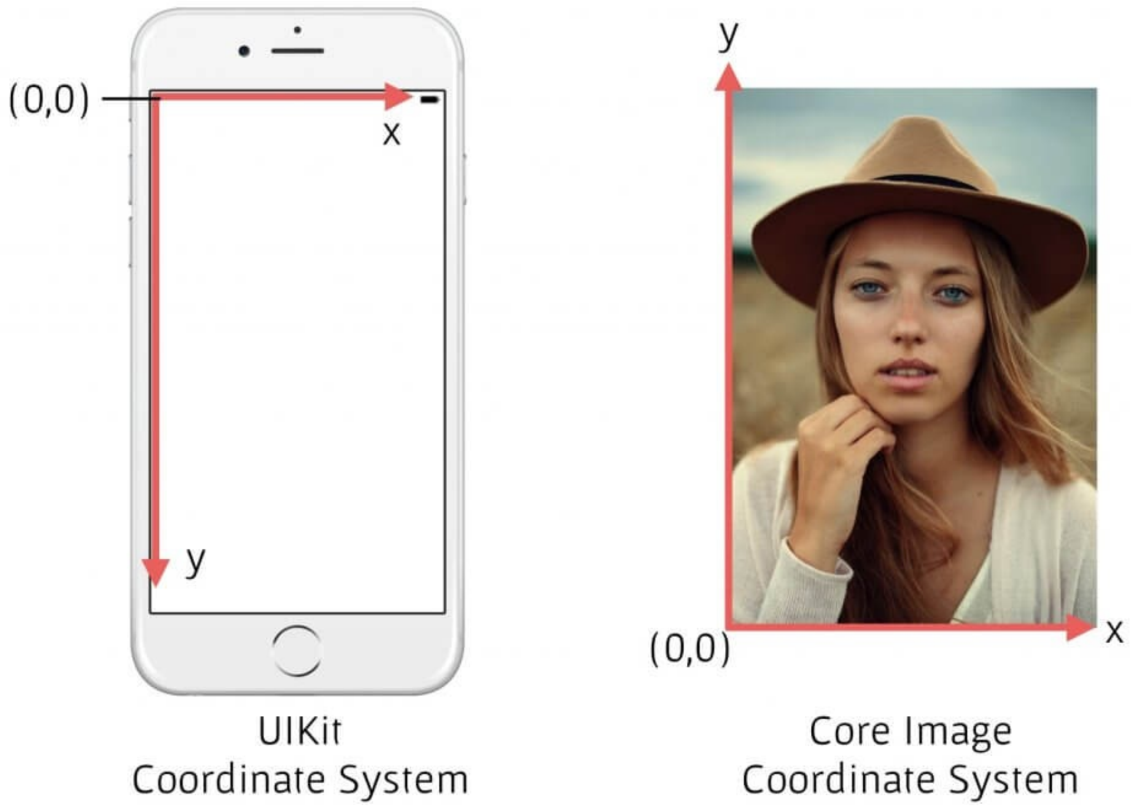


根据控制台的输出来看，貌似识别器识别到了人脸:

```
Found bounds are (177.0, 415.0, 380.0, 380.0)
```

当前的实现中没有解决的问题:

- 人脸识别是在原始图像上进行的，由于原始图像的分辨率比image view要高，因此需要设置image view的content mode为aspect fit(保持纵横比的情况下缩放图片)。为了合适的绘制矩形框，需要计算image view中人脸的实际位置与尺寸
- 还要注意的是，Core Image与UIView使用两种不同的坐标系统(看下表)，因此要实现一个CoreImage坐标到UIView坐标的转换。



现在使用下面的代码替换detect()方法:

```
func detect() {

    guard let personciImage = CIImage(image: personPic.image!) else {
        return
    }

    let accuracy = [CIDetectorAccuracy: CIDetectorAccuracyHigh]
    let faceDetector = CIDetector(ofType: CIDetectorTypeFace, context: nil, options: a
ccuracy)
    let faces = faceDetector.featuresInImage(personciImage)

    // For converting the Core Image Coordinates to UIView Coordinates
    let ciImageSize = personciImage.extent.size
    var transform = CGAffineTransformMakeScale(1, -1)
    transform = CGAffineTransformTranslate(transform, 0, -ciImageSize.height)

    for face in faces as! [CIFaceFeature] {

        print("Found bounds are \(face.bounds)")

        // Apply the transform to convert the coordinates
        var faceViewBounds = CGRectApplyAffineTransform(face.bounds, transform)

        // Calculate the actual position and size of the rectangle in the image view
        let viewSize = personPic.bounds.size
        let scale = min(viewSize.width / ciImageSize.width,
                        viewSize.height / ciImageSize.height)
        let offsetX = (viewSize.width - ciImageSize.width * scale) / 2
        let offsetY = (viewSize.height - ciImageSize.height * scale) / 2

        faceViewBounds = CGRectApplyAffineTransform(faceViewBounds, CGAffineTransformM
akeScale(scale, scale))
        faceViewBounds.origin.x += offsetX
        faceViewBounds.origin.y += offsetY

        let faceBox = UIView(frame: faceViewBounds)

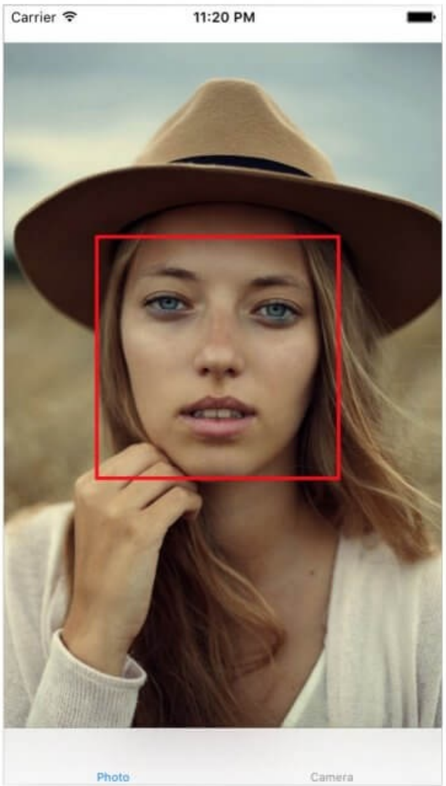
        faceBox.layer.borderWidth = 3
        faceBox.layer.borderColor = UIColor.redColor().CGColor
        faceBox.backgroundColor = UIColor.clearColor()
        personPic.addSubview(faceBox)

        if face.hasLeftEyePosition {
            print("Left eye bounds are \(face.leftEyePosition)")
        }

        if face.hasRightEyePosition {
            print("Right eye bounds are \(face.rightEyePosition)")
        }
    }
}
```

上述代码中，首先使用仿射变换将Core Image坐标转换为UIKit坐标，然后编写了计算实际位置与矩形视图尺寸的代码。

再次运行app，应该会看到人的面部周围会有一个框，干得漂亮！你已经成功使用Core Image识别出了人脸。



构建一个人脸识别的相机应用

想象一下你有一个用来照相的相机app，照完相后你想运行一下人脸识别来检测一下是否存在人脸。若存在一些人脸，你也许想用一些标签来对这些照片进行分类。我们不会构建一个保存照片后再处理的app，而是一个实时的相机app，因此需要整合一下UIImagePicker类，在照完相时立刻进行人脸识别。

在开始工程中已经创建好了CameraViewController类，使用如下代码实现相机的功能:

```
class CameraViewController: UIViewController, UIImagePickerControllerDelegate, UINavig
ationControllerDelegate {
    @IBOutlet var imageView: UIImageView!
    let imagePicker = UIImagePickerController()

    override func viewDidLoad() {
        super.viewDidLoad()

        imagePicker.delegate = self
    }

    @IBAction func takePhoto(sender: AnyObject) {

        if !UIImagePickerController.isSourceTypeAvailable(.Camera) {
            return
        }

        imagePicker.allowsEditing = false
        imagePicker.sourceType = .Camera

        presentViewController(imagePicker, animated: true, completion: nil)
    }

    func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaW
ithInfo info: [String : AnyObject]) {
        if let pickedImage = info[UIImagePickerControllerOriginalImage] as? UIImage {
            imageView.contentMode = .ScaleAspectFit
            imageView.image = pickedImage
        }

        dismissViewControllerAnimated(true, completion: nil)
        self.detect()
    }

    func imagePickerControllerDidCancel(picker: UIImagePickerController) {
        dismissViewControllerAnimated(true, completion: nil)
    }

}
```

前面几行设置UIImagePicker委托为当前视图类，在didFinishPickingMediaWithInfo方法(UIImagePicker的委托方法)中设置imageView为在方法中所选择的图像，接着返回上一视图调用detect函数。

还没有实现detect函数，插入下面代码并分析一下:

```
func detect() {
        let imageOptions =  NSDictionary(object: NSNumber(int: 5) as NSNumber, forKey:
CIDetectorImageOrientation as NSString)
        let personciImage = CIImage(CGImage: imageView.image!.CGImage!)
        let accuracy = [CIDetectorAccuracy: CIDetectorAccuracyHigh]
        let faceDetector = CIDetector(ofType: CIDetectorTypeFace, context: nil, option
s: accuracy)
        let faces = faceDetector.featuresInImage(personciImage, options: imageOptions
as? [String : AnyObject])

        if let face = faces.first as? CIFaceFeature {
            print("found bounds are \(face.bounds)")

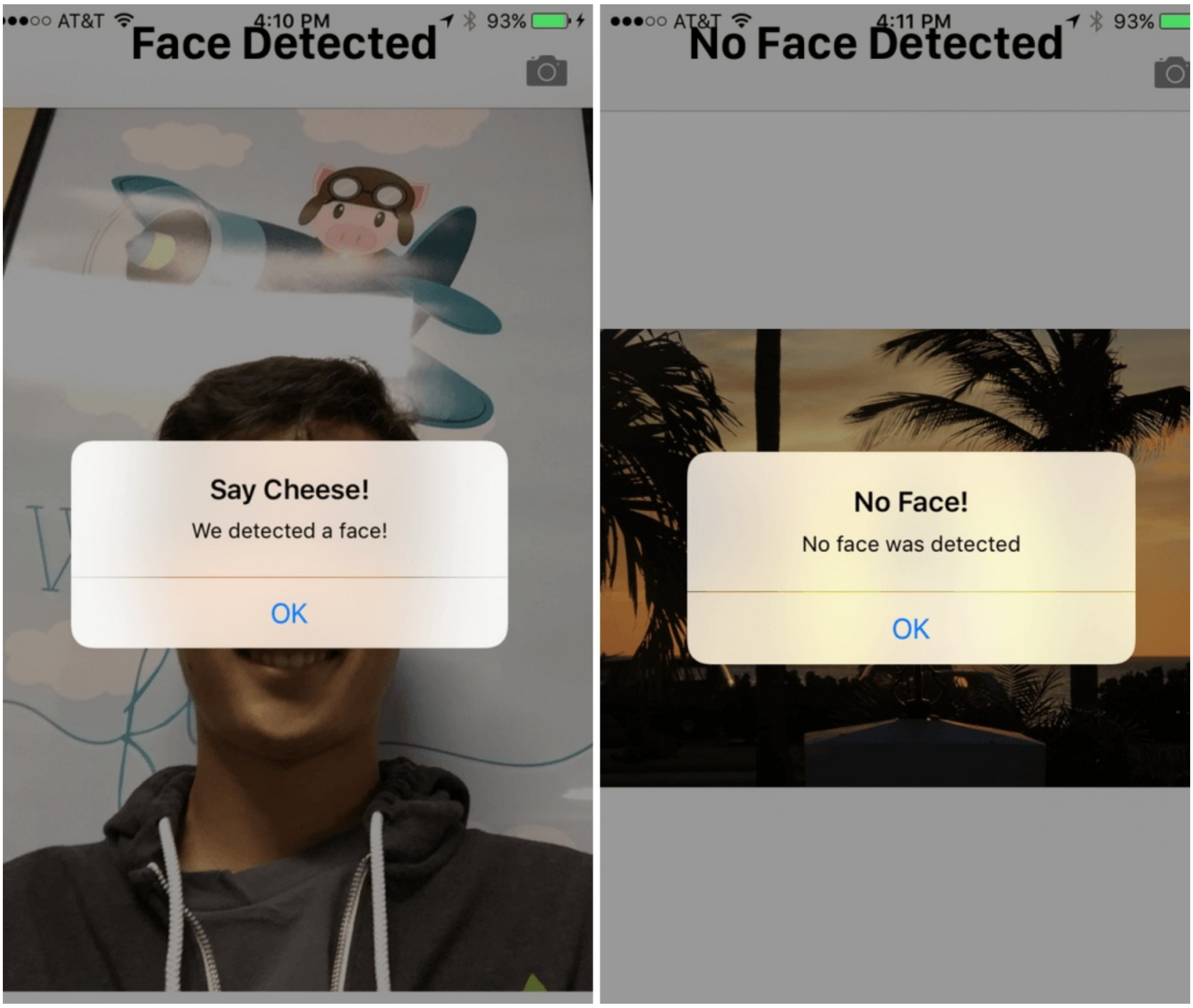
            let alert = UIAlertController(title: "Say Cheese!", message: "We detected
a face!", preferredStyle: UIAlertControllerStyle.Alert)
            alert.addAction(UIAlertAction(title: "OK", style: UIAlertActionStyle.Defau
lt, handler: nil))
            self.presentViewController(alert, animated: true, completion: nil)

            if face.hasSmile {
                print("face is smiling");
            }

            if face.hasLeftEyePosition {
                print("Left eye bounds are \(face.leftEyePosition)")
            }

            if face.hasRightEyePosition {
                print("Right eye bounds are \(face.rightEyePosition)")
            }
        } else {
            let alert = UIAlertController(title: "No Face!", message: "No face was det
ected", preferredStyle: UIAlertControllerStyle.Alert)
            alert.addAction(UIAlertAction(title: "OK", style: UIAlertActionStyle.Defau
lt, handler: nil))
            self.presentViewController(alert, animated: true, completion: nil)
        }
    }
```

这个detect()函数与之前实现的detect函数非常像，不过这次只用它来获取图像不做变换。当识别到人脸后显示一个警告信息“检测到了人脸！”，否则显示“未检测到人脸”。运行app测试一下



我们已经使用到了一些CIFaceFeature的属性与方法，比如，若想检测人物是否微笑，可以调用.hasSmile，它会返回一个布尔值。可以分别使用.hasLeftEyePosition与.hasRightEyePosition检测是否存在左右眼。

同样，可以调用hasMouthPosition来检测是否存在嘴，若存在则可以使用mouthPosition属性，如下所示:

```
if (face.hasMouthPosition) {
    print("mouth detected")
}
```

如你所见，使用Core Image来检测面部特征是非常简单的。除了检测嘴、笑容、眼睛外，也可以调用leftEyeClosed与rightEyeClosed检测左右眼是否睁开。

总结

在这篇教程中尝试了Core Image的人脸识别API与如何在一个相机app中应用它，构建了一个简单的UIImagePicker来选取照片并检测图像中是否存在人物。

如你所见，Core Image的人脸识别是个强大的API！希望这篇教程能给你提供一些关于这个鲜为人知的iOS API有用的信息。

在这里下载最终工程。

Unit Testing on macOS: Part 1

#

> - 原文链接 : [Unit Testing on macOS: Part 1/2]
> - 原文作者 : [Sarah Reichelt]
> - 译者 :

单元测试就是，我们心里都知道应该去做，但是却感觉它是繁杂无趣而且工作量巨大的一件麻烦事儿。

写代码如此愉悦，为啥还有人愿意花费写代码大半的时间来做代码检查？

理由就是自信！这在这个macOS上的单元测试入门教程中，你会学会如何测试你写的代码，还能获得自信，自信让代码按你的意愿去做事，自信即使对代码进行大改动也不会破坏原有的代码。

入门

示例工程使用Swift 3，同时必须使用至少是Xcode 8 beta 6的版本。下载初始工程并在Xcode中打开。

如果你看过raywenderlich.com中的任何一个教程，在这步你都想去构建运行程序了。但这次不同--你得去测试。打开**Product**菜单，选择**Test** ，提示你一个快捷键-- **Command-U** -- 之后你会多次用到这个快捷键。

当运行测试的时候，Xcode会先构建应用程序，然后应用程序的窗口出现几次之后，会看到消息提示“Test Succeeded”。在左侧的**Navigator**(导航栏)中，选中**Test Navigator**

这里显示三个默认加入的测试栏目；每个测试栏目的边上都有一个绿色的小对勾，表示测试已经通过。如果想看测试栏目都包含了哪些文件，点击**Test Navigator**的里面第二行有大写T字母图标开头的叫**High Roller Tests**的栏目。

有几个要点需要提示一下:

- import引用: **XCTest**是Xcode提供的测试框架。 `@testable import High_Roller` 引入是用来给测试代码访问 `High_Roller` 模块用的。每个测试文件都需要引入这二个。
- `setup()` 和 `tearDown()` : 这二个方法会分别在每个单独的测试方法调用前后被调用。
- `testExample()` 和 `testPerformanceExample()` :真正的测试部分。第一个方法用来测试代码功能，第二个用来测试代码性能。每个测试方法都必须以test开头，这样Xcode才能区分出这些代码是用于执行测试的。

单元测试是什么

在你开始写单元测试之前，有必要简要了解一下什么是单元测试以及我们为什么要用单元测试。

单元测试就是测试单独的代码片段或者一组代码片段的一个函数。它并不会被包含到你开始的应用程序当中，只是在开发阶段用来测试你所写的代码是否按你的期望来运行的。

大部分人对单元测试的第一反应是："你让我写二份代码？一份是应用程序自己内部的功能代码，另一份是测试这个功能的代码？"，但事实可能比二份还要多--一些大的工程到最后的时候测试代码比功能本身的代码还要多得多。

首先这看起来非常耗费时间和精力--但是一旦你通过测试发现了你代码中不曾注意到的错误，或者发觉了重构代码产生的副作用，你就会体会到单元测试是个多么棒的工具。任何没有单元测试的工程在用不了多久之后就会显得很不靠谱，你不太敢对工程做改动，因为你不知道改动之后会发生什么。

测试驱动开发

测试驱动开发(TDD)是单元测试的一个分支，就是只针对必须的测试步骤编写对应的代码。

Unit Testing on macOS: Part 2

#

> - 原文链接：[Unit Testing on macOS: Part 2/2]
> - 原文作者：[Sarah Reichelt]
> - 译者：

Geofencing Tutorial with Core Location

Core Location的地理围栏开发指南

> - 原文链接：Geofencing Tutorial with Core Location
> - 原文作者：Andy Pereira
> - 译者：yrq110

> 更新: 这篇教程已更新为 Xcode 8 / Swift 3。

来创建围栏吧!

当设备进入或离开你所设置的区域时，地理围栏会给app发送一条通知提醒你。这可以让你做一些很cool的事：比如设置一个当你离开家时的通知，或者当附近有用户喜爱的商店时用最新与最棒的商品来迎接用户。在这篇地理围栏的教程中，会学到如何在iOS中使用Swift进行区域监测 - 使用Core Location中的Region Monitoring API。

在这篇教程中会创建一个名为Geotify的位置提醒应用，让用户创建提示器并与真实世界的实际位置相关联。开始吧!

入门

下载开始工程。这个工程提供一个简单的功能：允许用户在地图上添加或删除标注点，每个标注点都表示一个位置提醒，或者可以叫它——地理通知(PS:geotification，自造词)。

构建并运行，会看到如下这样一个空的map view。

点击导航栏的+按钮添加一个新的地理通知。app会出现一个独立的视图来设置地理通知的各种属性。

在这篇教程中需要在库比蒂诺的苹果总部添加一个大头针，如何你不知道在哪，打开google地图去找到这个点，记得活用缩放功能提高准确度！

> 注意：在模拟器中使用捏合手势进行缩放，按住option键的同时按下shift移动捏合中心，然后松开shift单击拖动进行捏合操作。

radius表示与指定位置间的距离，超过这个距离会触发iOS的通知，note表示显示在通知中的信息。app也允许用户使用顶部的分段控件来设置通知的触发时机是进入还是离开圆形的地理围栏时。

在radius中输入1000，note中输入Say Hi to Tim!顶部选择Upon Entry，这样，第一次地理通知就设置好了。

点击Add添加，会看到在map view中出现了一个新标记的大头针，就是你刚刚添加的地理通知，标记点附近有一个圆圈表示定义的地理围栏:

点击大头针会显示地理通知的详细信息，比如提醒的内容与之前指定的事件类型。别点那个小叉号除非你想删掉它!

可以随意添加或删除任何地理通知。由于app使用NSUserDefaults存储持久化数据，因此重启app时会保留之前的地理通知列表。

设置LocationManager与许可

至此，你添加在map view上的任何地理通知都仅仅是显示了出来，并没有实际的检测效果。为了实现围栏检测的功能，需要在Core Location上注册每个与地理通知关联的围栏。

在启用围栏监测前，需要设置一个LocationManager实例并且请求到一些许可。

打开GeotificationsViewController.swift在类顶部声明一个CLLocationManager的常量实例，如下:

```
class GeotificationsViewController: UIViewController {

  @IBOutlet weak var mapView: MKMapView!

  var geotifications = [Geotification]()
  let locationManager = CLLocationManager() // Add this statement

  ...
}
```

接着使用如下代码替换viewDidLoad()方法:

```
override func viewDidLoad() {
  super.viewDidLoad()
  // 1
  locationManager.delegate = self
  // 2
  locationManager.requestAlwaysAuthorization()
  // 3
  loadAllGeotifications()
}
```

来一步步实现这个方法:

1. 将locationManager实例的委托设为当前视图控制器。
2. 调用requestAlwaysAuthorization()方法，提示用户程序请求一个使用地理位置服务的授权。app的地理围栏功能需要Always authorization(前后台都能使用的授权), 这是因为围栏检测在app未在前台运行时也要进行检测。Info.plist中的NSLocationAlwaysUsageDescription键已经设置好了当请求用户位置时所显示的信息。
3. 调用loadAllGeotifications(),反序列化之前在NSUserDefaults中保存的地理通知列表，并将其保存在本地的地理通知数组中。这个方法也会加载map view上标注的地理通知。

当app提示用户授权时，会显示NSLocationAlwaysUsageDescription中的值，友好的说明为何app需要访问用户的地理位置。这个键是请求位置服务授权时所必须要设置的，若没有这个键，则系统会忽略请求并拒绝提供位置服务。

构建并运行，会看到一个用户提示，显示的信息为之前所设置的:

已完成了设置app请求许可的部分，干得好！点击Allow确保location manager在适当的时候能接收到委托方法的回调。

在实现地理围栏之前，还有一个小问题需要解决：当前的用户位置还未显示在map view上！这个特性还未启用，导致导航栏左上的缩放按钮不起作用。

还好这个问题不是什么大事，需要在app获得授权后启用map view显示当前位置的功能。

进入GeotificationsViewController.swift文件，在CLLocationManagerDelegate扩展中添加如下委托方法，:

```
extension GeotificationsViewController: CLLocationManagerDelegate {
  func locationManager(_ manager: CLLocationManager, didChangeAuthorization status: CL
AuthorizationStatus) {
    mapView.showsUserLocation = (status == .authorizedAlways)
  }
}
```

当授权状态改变时，location manager就会调用locationManager(_:didChangeAuthorizationStatus:)方法。若用户已经给予了app使用位置服务的许可，则这个方法会在初始化location manager并设置它的委托后被调用。

这个方法放在了合适的地方，用来检测app是否被授权，若被授权，则在map view中显示用户的当前位置。

构建并运行app，若是在真机上运行的，则会看到主map view中会出现一个位置标记。若是在模拟器上运行的，点击菜单中的Debug\Location\Apple来显示位置标记:

另外，导航栏上的缩放按钮现在起作用了。:]

注册地理围栏

在正常设置完location manager后，接下来需要做的工作是在app中注册用户的地理围栏。

在app中，用户的地理围栏信息是使用自定义的地理通知模型存放的。不过，Core Location要求每个地理围栏在注册前都必须是一个CLCircularRegion实例。为了解决这个问题，需要创建一个帮助方法来将地理通知对象转换为CLCircularRegion。

打开GeotificationsViewController.swift文件，添加如下方法:

```
func region(withGeotification geotification: Geotification) -> CLCircularRegion {
  // 1
  let region = CLCircularRegion(center: geotification.coordinate, radius: geotificatio
n.radius, identifier: geotification.identifier)
  // 2
  region.notifyOnEntry = (geotification.eventType == .onEntry)
  region.notifyOnExit = !region.notifyOnEntry
  return region
}
```

上面的代码中做了哪些事情:

1. 首先使用地理围栏的位置、半径与一个识别器(为了使iOS区分app中注册的围栏)初始化了一个CLCircularRegion。初始化是很简单的，Geotification模型已经包含了所需要的属性。
2. CLCircularRegion实例有2个布尔型属性：notifyOnEntry和notifyOnExit。这些标识指定了当设备进入或离开指定围栏时是否触发围栏事件。若app设计为对每个围栏仅允许一个通知，根据Geotification对象中保存的枚举值，将其中一个标识设为true另一个设为false即可。

接着，需要一个当用户添加地理通知时启动监测的方法。

在GeotificationsViewController中添加如下方法:

```
func startMonitoring(geotification: Geotification) {
  // 1
  if !CLLocationManager.isMonitoringAvailable(for: CLCircularRegion.self) {
    showAlert(withTitle:"Error", message: "Geofencing is not supported on this device!"
)
    return
  }
  // 2
  if CLLocationManager.authorizationStatus() != .authorizedAlways {
    showAlert(withTitle:"Warning", message: "Your geotification is saved but will only
 be activated once you grant Geotify permission to access the device location.")
  }
  // 3
  let region = self.region(withGeotification: geotification)
  // 4
  locationManager.startMonitoring(for: region)
}
```

来逐步分析下:

1. isMonitoringAvailableForClass(_:)表示设备是否含有进行围栏检测所需的硬件。若监测功能不可用则显示一个合适的警告。showSimpleAlertWithTitle(_:message:viewController)是一个Utilities.swift中的辅助函数，接收标题与消息数据，弹出一个警告视图。
2. 接着，需要检查授权状态，确保app得到使用位置服务所需的许可。若app未被授权，则无法接受到任何围栏相关的通知。 不过，即使在app未被授权的情况下，也要允许用户保存地理通知，Core Location是允许你注册围栏的。在用户在之后给app授权时，自动启用这些围栏的监测功能。
3. 使用输入的地理通知与之前定义的帮助方法创建一个CLCircularRegion实例。
4. 最后，使用Core Location注册CLCircularRegion实例进行监测。

在启用监测的方法搞定后，也需要一个当用户移除地理通知时停止监测的方法。

在GeotificationsViewController.swift文件中，在startMonitoringGeotificiation(_:)下面添加如下方法:

```
func stopMonitoring(geotification: Geotification) {
  for region in locationManager.monitoredRegions {
    guard let circularRegion = region as? CLCircularRegion, circularRegion.identifier
== geotification.identifier else { continue }
    locationManager.stopMonitoring(for: circularRegion)
  }
}
```

这个方法命令locationManager停止监测与输入的地理通知关联的CLCircularRegion。

好了，现在完成启动和停止的方法了，会在添加或移除地理通知时使用这些方法，是时候进行添加通知部分的开发了。

首先，看看GeotificationsViewController.swift文件中的addGeotificationViewController(_:didAddCoordinate)方法。

这个方法是由AddGeotificationViewController调用的委托方法来创建地理通知，这个方法负责使用来自AddGeotificationsViewController的值创建一个新的Geotification对象，并且更新对应的map view与地理通知列表。接着调用saveAllGeotifications()，更新地理通知列表，并保存到NSUserDefaults中。

使用如下代码替换这个方法:

```
func addGeotificationViewController(controller: AddGeotificationViewController, didAdd
Coordinate coordinate: CLLocationCoordinate2D, radius: Double, identifier: String, not
e: String, eventType: EventType) {
  controller.dismiss(animated: true, completion: nil)
  // 1
  let clampedRadius = min(radius, locationManager.maximumRegionMonitoringDistance)
  let geotification = Geotification(coordinate: coordinate, radius: clampedRadius, ide
ntifier: identifier, note: note, eventType: eventType)
  add(geotification: geotification)
  // 2
  startMonitoring(geotification: geotification)
  saveAllGeotifications()
}
```

代码中的两个关键点:

1. 确保半径值与locationManager的maximumRegionMonitoringDistance属性一致，这个属性被定义为地理围栏的最大半径。这很重要，任何超过这个最大值的值都会使监测失败。
2. 调用startMonitoringGeotification(_:)，使用Core Location注册与新添加的地理通知相关联的监测围栏。

现在，在app中已经可以注册新的监测围栏了，不过有一个限制：地理围栏是一个共享的系统资源，Core Location对于每个app限制注册的围栏数量最多为20个。

虽然对于这个限制有可变通的方法，对于本教程来说，限制一下用户添加的地理通知数量即可。

如下，在updateGeotificationsCount()方法中添加一行代码:

```
func updateGeotificationsCount() {
  title = "Geotifications (\(geotifications.count))"
  navigationItem.rightBarButtonItem?.isEnabled = (geotifications.count < 20)  // 添加这
一行
}
```

当app的通知数量达到临界值时禁用导航栏上的添加按钮。

最后，来处理一下地理通知的移除操作，这个操作在mapView(_:annotationView:calloutAccessoryControlTapped:)方法中进行，当用户点击每个标注点附带的delete按钮时会调用这个方法。

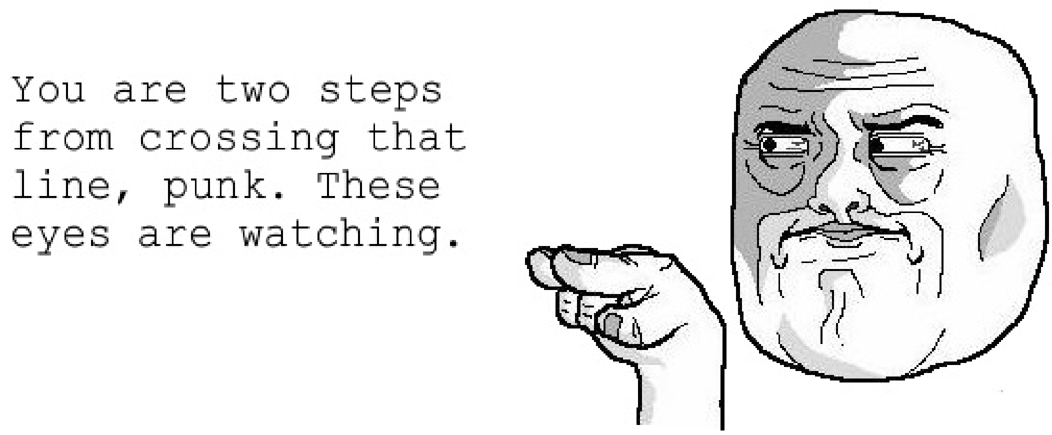
在mapView(_:annotationView:calloutAccessoryControlTapped:)方法中添加调用stopMonitoring(geotification:)的代码:

```
func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView, calloutAcces
soryControlTapped control: UIControl) {
  // 删除地理通知
  let geotification = view.annotation as! Geotification
  stopMonitoring(geotification: geotification)   // 添加的语句
  removeGeotification(geotification)
  saveAllGeotifications()
}
```

添加的语句会停止与地理通知相关围栏的监测工作，接着移除通知并将改动保存到NSUserDefaults中。

好了，现在app支持监测与取消监测用户的地理围栏，加油！

构建并运行工程，虽然看不到明显的变化不过app现在可以注册可监测的地理围栏了。然而还不能响应任何围栏事件，不用担心，这是接下来的任务！



添加地理围栏响应事件

首先来实现一些处理错误的委托方法--这很重要，以免发生一些问题后不知何因。

进入GeotificationsViewController.swift文件, 在CLLocationManagerDelegate的扩展中添加如下方法:

```
func locationManager(_ manager: CLLocationManager, monitoringDidFailFor region: CLRegi
on?, withError error: Error) {
  print("Monitoring failed for region with identifier: \(region!.identifier)")
}

func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
  print("Location Manager failed with the following error: \(error)")
}
```

这几个委托方法仅仅以日志形式输出了location manager遇到的错误，方便调试。

> 注意: 你肯定会想在生产环境中的app更有效的处理这些错误，而不是使其悄无声息的出错，比如可以提示用户哪里出了问题。

接着，打开AppDelegate.swift，在这里添加监听与响应地理围栏的进入与离开事件的代码。

在文件头部引入CoreLocation框架，添加如下代码:

```
import CoreLocation
```

确保AppDelegate中包含一个CLLocationManager实例:

```
class AppDelegate: UIResponder, UIApplicationDelegate {
  var window: UIWindow?

  let locationManager = CLLocationManager() // Add this statement
  ...
}
```

使用如下代码替换application(_:didFinishLaunchingWithOptions:):

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOpt
ions: [UIApplicationLaunchOptionsKey : Any]? = nil) -> Bool {
  locationManager.delegate = self
  locationManager.requestAlwaysAuthorization()
  return true
}
```

这样就设置了在AppDelegate中接收与围栏相关的事件，不过你也许会想“为何我不让view controller而是AppDelegate来做这件事？”

app中注册的围栏一直处于被监测的状态，即使在app未运行时。若设备在app未运行的情况下触发了一个围栏事件，则iOS会自动从后台重启app。这使得AppDelegate是一个用来处理事件的理想入口，因为此时的view controller有可能还未加载或未准备好。

你也许还会想“一个新建的CLLocationManager实例是如何获知所监测的围栏的？”

实际上所有app中所注册的监测用围栏可以轻松地被所有location managers访问，因此location managers的初始化位置就无所谓了。挺棒的，不是吗？:]

现在剩下的就是实现响应围栏事件的相关委托方法了。在这么做之前，需要添加一个处理围栏事件的函数。

在AppDelegate.swift中添加如下方法:

```
func handleEvent(forRegion region: CLRegion!) {
  print("Geofence triggered!")
}
```

这个方法接收一个CLRegion对象并简单的输出一条文本信息。别担心-之后会实现具体的细节。

接下来，在AppDelegate.swift中的CLLocationManagerDelegate扩展下添加如下委托方法，并添加一个刚刚创建的handleRegionEvent(_:) 函数的调用:

```
extension AppDelegate: CLLocationManagerDelegate {

  func locationManager(_ manager: CLLocationManager, didEnterRegion region: CLRegion)
{
    if region is CLCircularRegion {
      handleEvent(forRegion: region)
    }
  }

  func locationManager(_ manager: CLLocationManager, didExitRegion region: CLRegion) {
    if region is CLCircularRegion {
      handleEvent(forRegion: region)
    }
  }
}
```

建议使用合适的名称来给方法起名，当设备进入一个CLRegion区域时启动locationManager(_:didEnterRegion:)，离开CLRegion区域时启动locationManager(_:didExitRegion:)。

这些方法都会返回CLRegion，需要检查并确保是否为CLCircularRegion对象，因为有可能是一个CLBeaconRegion对象若app监测的是iBeacon区域的话。若确实是一个CLCircularRegion对象，则调用handleRegionEvent(_:)。

> 注意: 围栏事件仅会在iOS设备跨越边界时才会被触发。若用户已经在注册的围栏中，则iOS不会产生事件。苹果提供了一个requestStateForRegion(_:)方法用来查询设备位置是在所给围栏的里面还是外面。

现在app可以接收到围栏事件了，是时候来测试一下了。:]

测试app时最准确的方法是将程序部署到你的设备上，添加一些地理通知，然后去行走或驾车测试。

不过现在最好不要那样做，现在还不能确认触发生产环境中的设备围栏事件时输出的日志信息，除此之外，在实际测试前保证app能正常工作也是极好的。

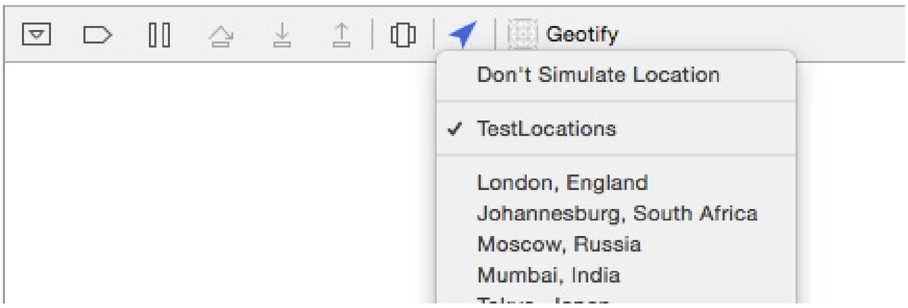
幸运的是，有一个不用离开舒适的家里就可以搞定的简单方法，Xcode允许在工程中使用一个硬编码的GPX文件来模拟测试位置。为了方便起见在开始工程中就已经包含这个文件了。:]

在Supporting Files文件夹中找到并打开TestLocations.gpx, 选择inspect它的内容，会看到如下所示的代码:

```
<?xml version="1.0"?>
<gpx version="1.1" creator="Xcode">
  <wpt lat="37.422" lon="-122.084058">
    <name>Google</name>
  </wpt>
  <wpt lat="37.3270145" lon="-122.0310273">
    <name>Apple</name>
  </wpt>
</gpx>
```

GPX文件实际上是一个包含下面两个基准点的XML文件：山景城的Google总部与库比蒂诺的苹果总部。

构建并运行工程，开始模拟GPX文件中的位置。当app启动主view controller后，回到Xcode中，选择Debug工具栏的Location图标，然后选择TestLocations:

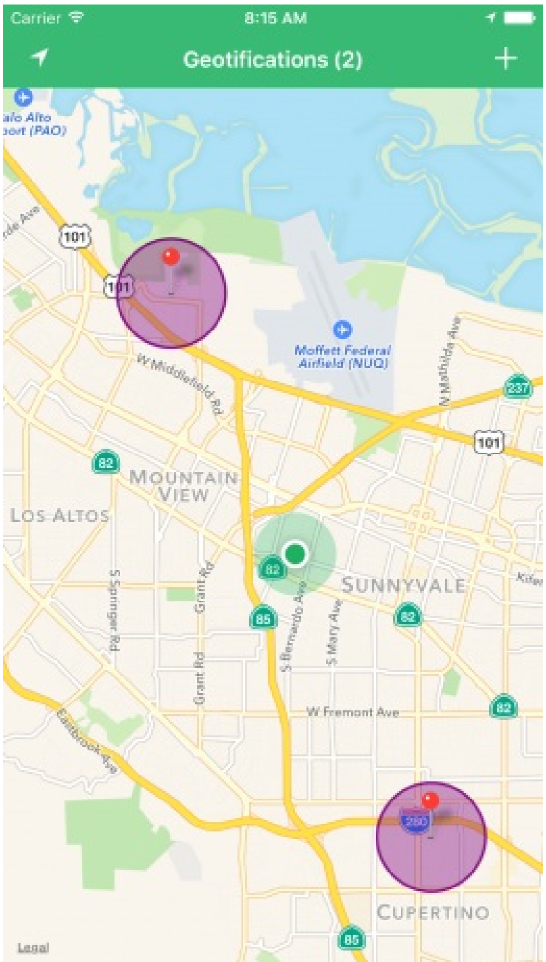


回到app, 使用导航栏左上的缩放按钮找到当前位置。接近这个区域时，会看到一个位置标记在Google与苹果总部间不断移动。

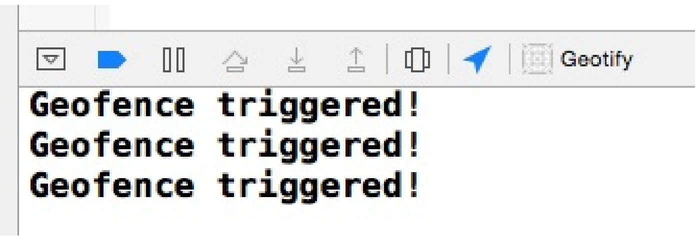
通过添加一些基准点路径上的地理通知来测试app。若在实现注册地理通知之前添加过一些通知，则这些通知是不起作用的，需要清除一下并重启。

对于测试的位置，一个好的点子是将地理通知放置在每个基准点上，可以使用下面的测试方案:

- Google: 半径: 1000m, 信息: “Say Bye to Google!”, 离开时通知
- Apple: 半径: 1000m, 信息: “Say Hi to Apple!”, 进入时通知



在添加通知后，每次当位置标记进入或离开一个围栏时都会在控制台输出一条日志。若按了home键或锁屏将app放置在了后台，当设备每次穿过围栏时同样能看到日志信息，虽然不能在屏幕中看到并确认移动的动作。



> 注意: 位置模拟可以在iOS模拟器中使用也可以在真机中使用。不过，在iOS模拟器中的精度较低，触发事件的时机与看到的模拟位置进出围栏的时刻不是很吻合。最好用自己的真机来进行位置模拟，或者直接出门走走试试！

添加围栏事件的通知

已经完成了app中的大部分内容了，现在剩下的工作就是实现当设备穿过一个地理通知的围栏时提醒用户的功能了。

为了获取与触发的CLCircularRegion关联的记录，需要检索NSUserDefaults中对应的地理通知。这不难，只要使用在注册时分配给CLCircularRegion的identifier就可以找到正确的地理通知。

进入AppDelegate.swift,在类的底部添加如下辅助方法:

```
func note(fromRegionIdentifier identifier: String) -> String? {
  let savedItems = UserDefaults.standard.array(forKey: PreferencesKeys.savedItems) as?
 [NSData]
  let geotifications = savedItems?.map { NSKeyedUnarchiver.unarchiveObject(with: $0 as
Data) as? Geotification }
  let index = geotifications?.index { $0?.identifier == identifier }
  return index != nil ? geotifications?[index!]?.note : nil
}
```

这个辅助方法会根据identifier从持久化数据中检索并返回地理通知。

现在可以检索到与围栏相关联的记录了，需要再编写一段代码，用于当围栏事件启动时发送一个通知，将记录作为显示的信息。

在application(_:didFinishLaunchingWithOptions:)方法的底部，return语句之前添加如下代码:

```
application.registerUserNotificationSettings(UIUserNotificationSettings(types: [.sound
, .alert, .badge], categories: nil))
UIApplication.shared.cancelAllLocalNotifications()
```

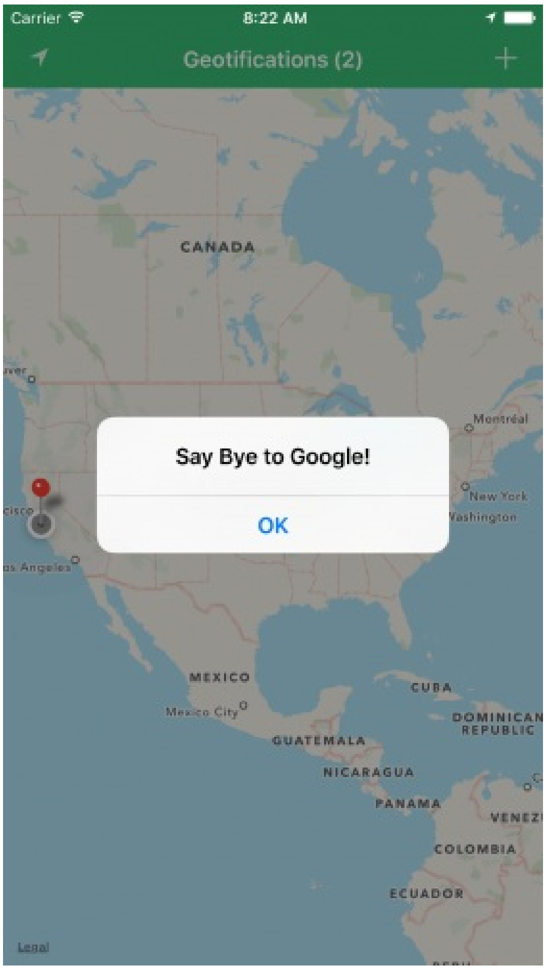
添加的代码用于使用户给予app发送通知的许可，并做了一下扫除-清楚已存在的通知。

接着，使用如下代码替换handleRegionEvent(_:):

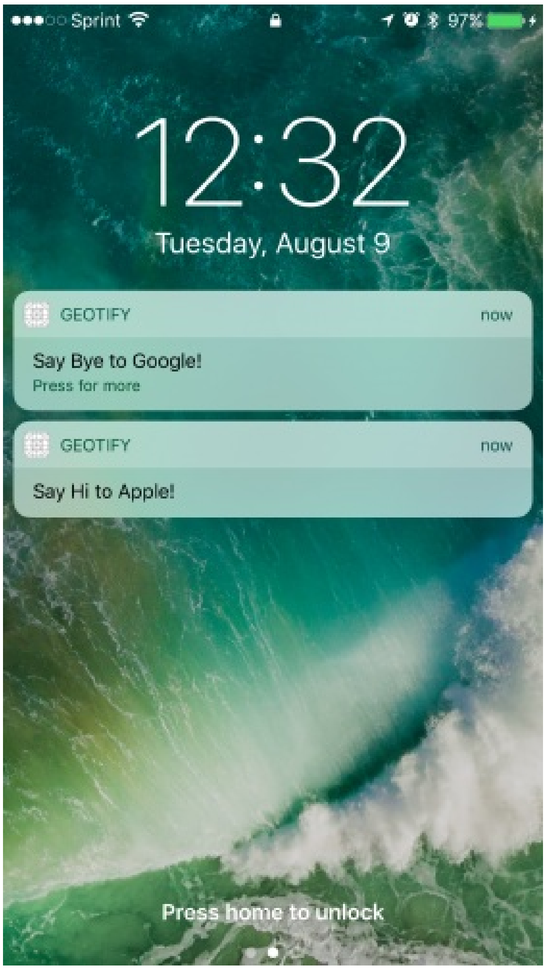
```
func handleEvent(forRegion region: CLRegion!) {
  // 若app正在运行则显示一个警告框
  if UIApplication.shared.applicationState == .active {
    guard let message = note(fromRegionIdentifier: region.identifier) else { return }
    window?.rootViewController?.showAlert(withTitle: nil, message: message)
  } else {
    // 否则显示一个本地通知
    let notification = UILocalNotification()
    notification.alertBody = note(fromRegionIdentifier: region.identifier)
    notification.soundName = "Default"
    UIApplication.shared.presentLocalNotificationNow(notification)
  }
}
```

若app正在运行，则会显示一个包含记录信息的警告视图，否则发送包含相同信息的本地通知。

构建并运行工程，在测试中触发围栏事件时，会看到一个警告视图，显示提醒信息:



在测试过程中按下Home键或锁屏将app置于后台，依然可以周期性的收到围栏事件的通知:



至此，已经亲手完成了一个功能全面的位置提醒app了。好了，出门试试app吧！

Building Your First Web App in Swift Using Vapor

使用Vapor构建你的第一个Swift Web应用

> - 原文链接：Building Your First Web App in Swift Using Vapor
> - 原文作者：SAHAND EDRISIAN
> - 译者：yrq110

苹果在WWDC 2015上宣布swift将会开源，很快，在2015年的12月Swift的源码就在GitHub上公开了。

Swift的开源给了开发者更多的机会与可能性，扩展Swift生态圈的应用。

如所期待的那样，开发者们很快就探索出了Swift的很多其他方面的应用，这篇教程中所介绍的web app就是其中的一种应用。

为何关注?

在服务端使用一种强类型的语言是有很多益处的。如果想给你的iOS app开发后端功能的话，使用同一种语言与风格比较容易保持一致性。

这里有三种很不错的Swift服务端：Perfect, Kitura与Vapor。这篇教程中会使用Vapor探索在Swift服务端上的神奇Swift世界。

我会指导你如何安装Vapor与Swift3，学习服务端Swift的基础知识，然后在Heroku上部署网站/后端。

前提

这篇教程中会使用很多bash命令，因此需要掌握一些基本的bash与终端知识。Vapor是基于Swift3的，因此也需要Xcode8。

Vapor与其他Swift工程(比如Swift包管理器)都是基于Swift3的，建议你通过这篇文章了解下Swift3中的一些变化。

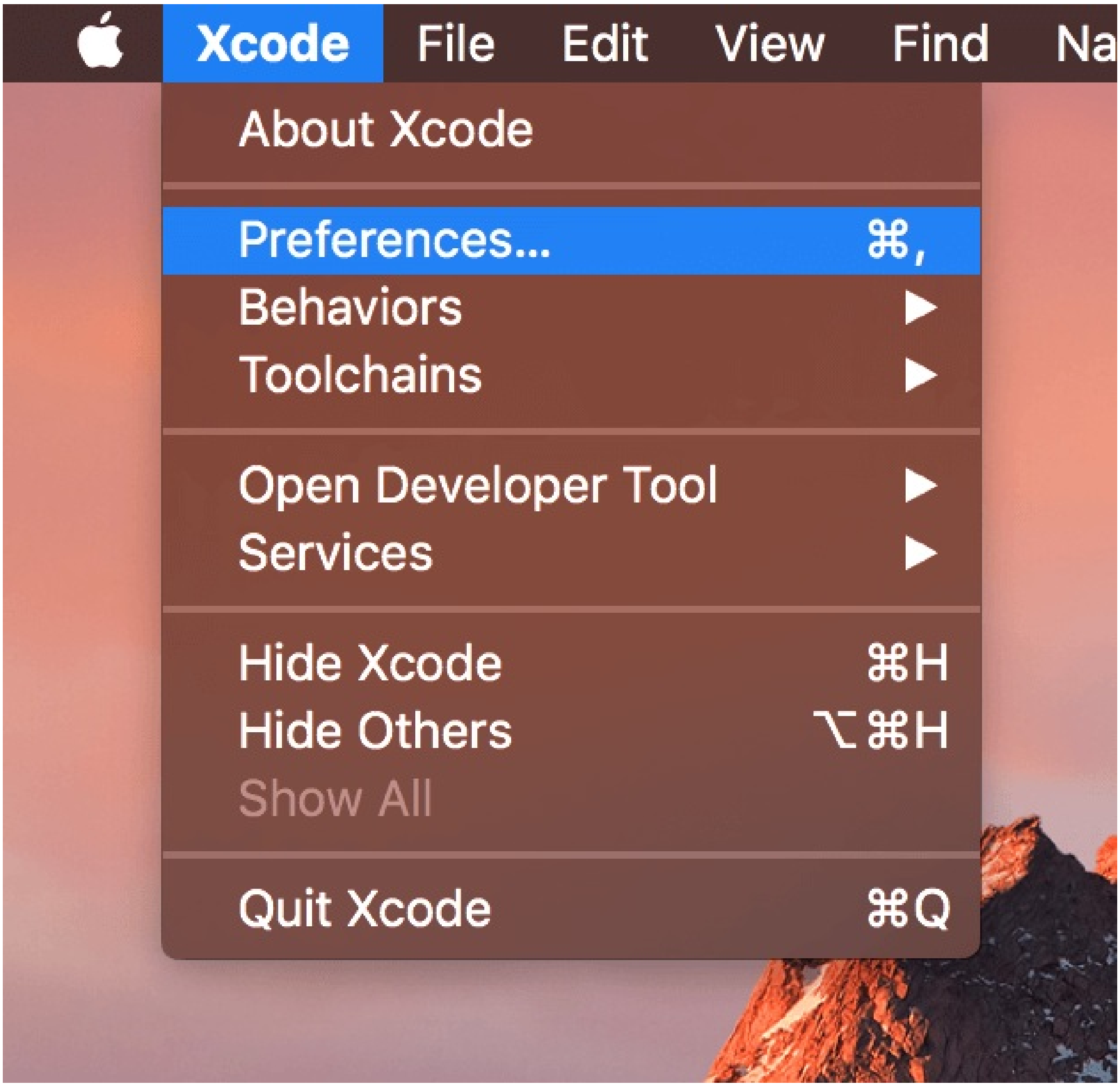
在教程的最后，会教你如何在Heroku上部署Vapor服务端，Heroku是个很火的云服务供应商，推荐你尝试下。

开搞吧!

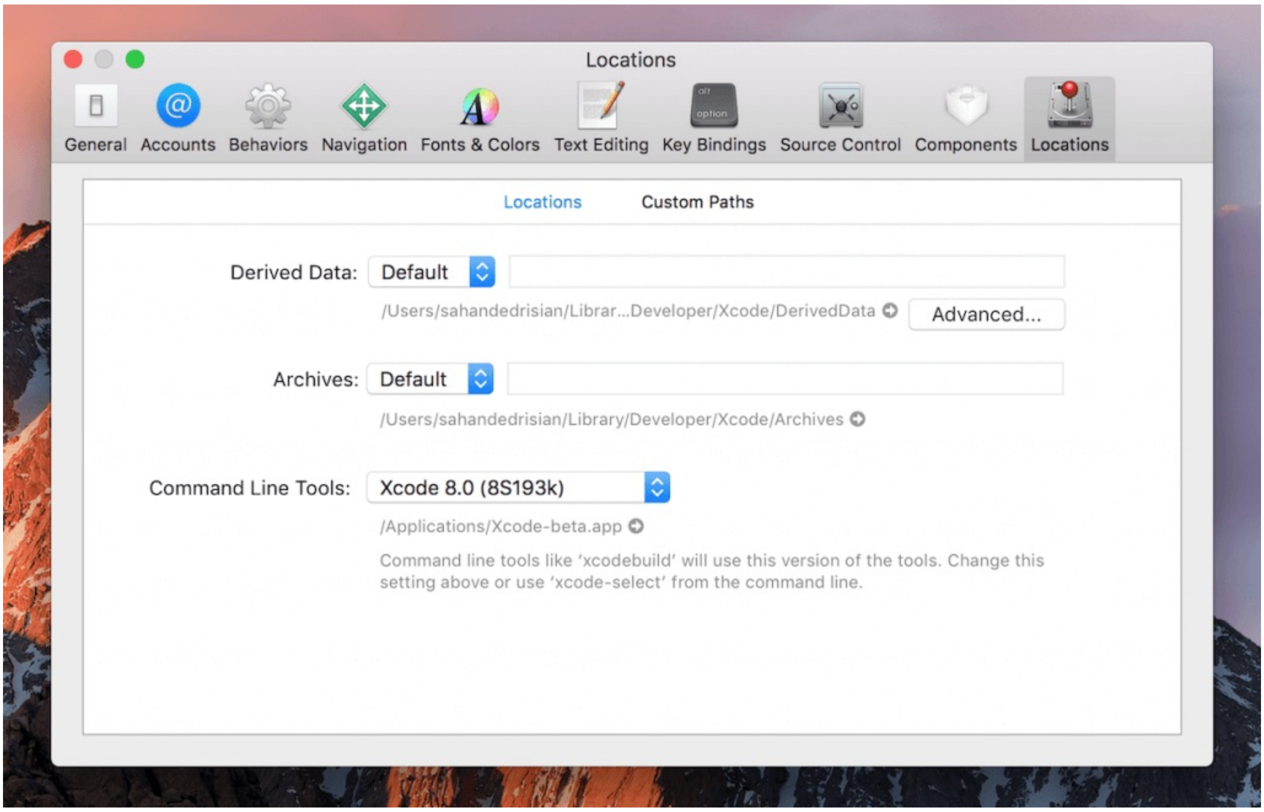
安装Vapor

Vapor需要Swift 3与Xcode8，可以在这里下载GM版Xcode。

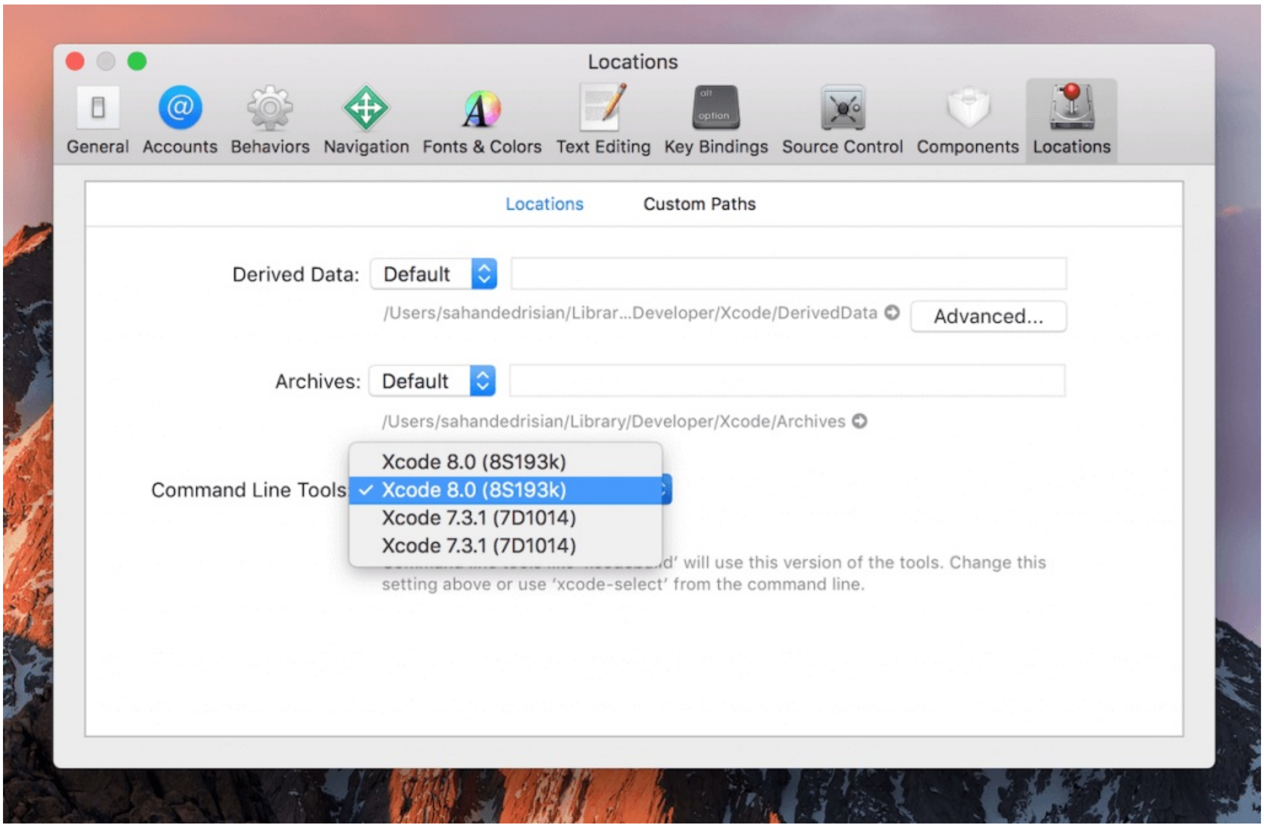
首先需要选择最新的命令行工具，打开Xcode的preferences(偏好设置)。



接着来到导航栏的Locations设置。



最后，选择最新的Xcode8命令行工具集。



安装Swiftenv

是时候安装Swift 3了。首先安装Swiftenv，使用Swiftenv可以很轻松的安装Swift，并且可以在多个版本间切换。通过clone Github上的仓库来安装Swiftenv，在终端里运行如下代码:

```
git clone https://github.com/kylef/swiftenv.git ~/.swiftenv
```

然后在bash文件中初始化Swiftenv，bash文件就是存放bash属性与一些设置的地方。在终端里使用如下命令打开bash文件:

```
open ~/.bash_profile
```

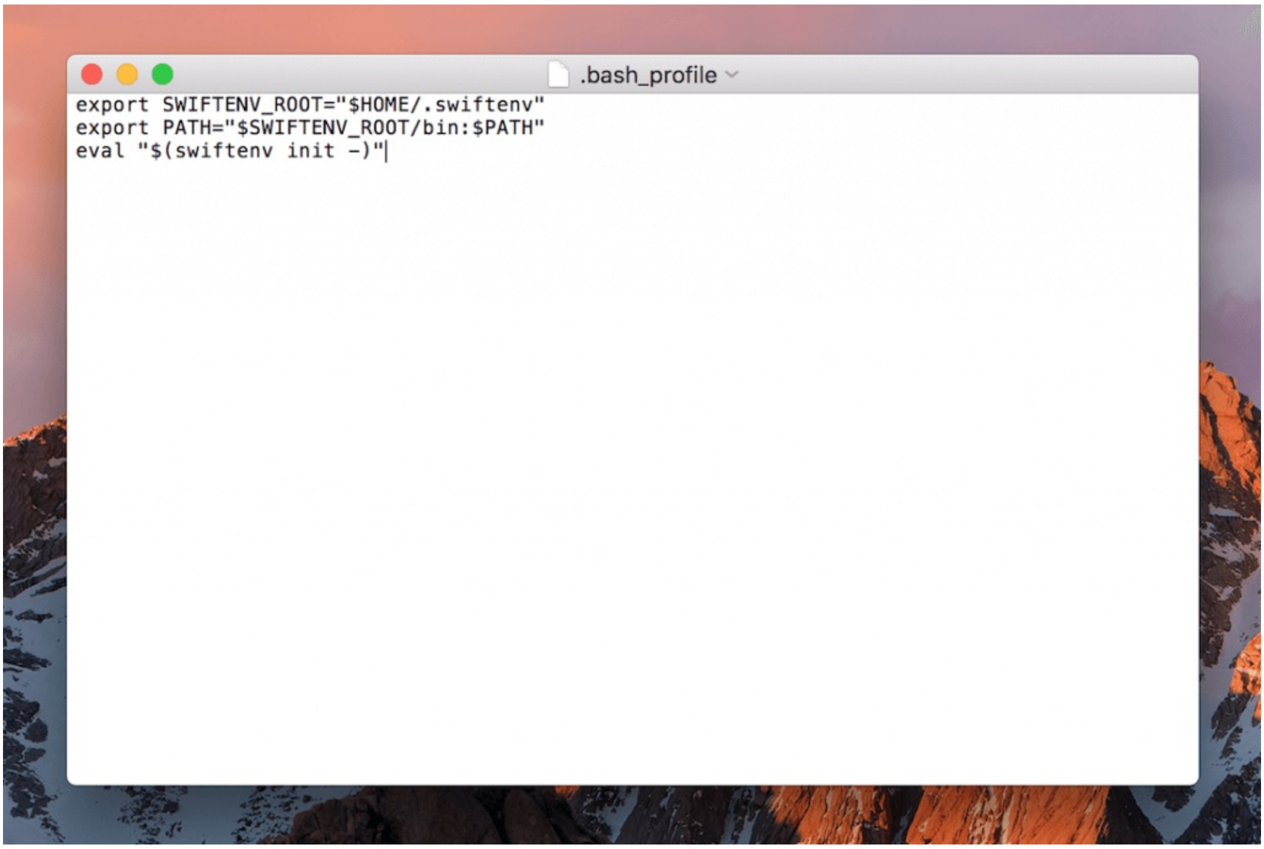
open指令会使用TextEdit打开你的bash文件，若没有bash文件，使用如下指令创建它:

```
touch ~/.bash_profile
```

打开bash文件后，复制如下命令，用来初始化Swiftenv。

```
export SWIFTENV_ROOT="$HOME/.swiftenv"
export PATH="$SWIFTENV_ROOT/bin:$PATH"
eval "$(swiftenv init -)"
```

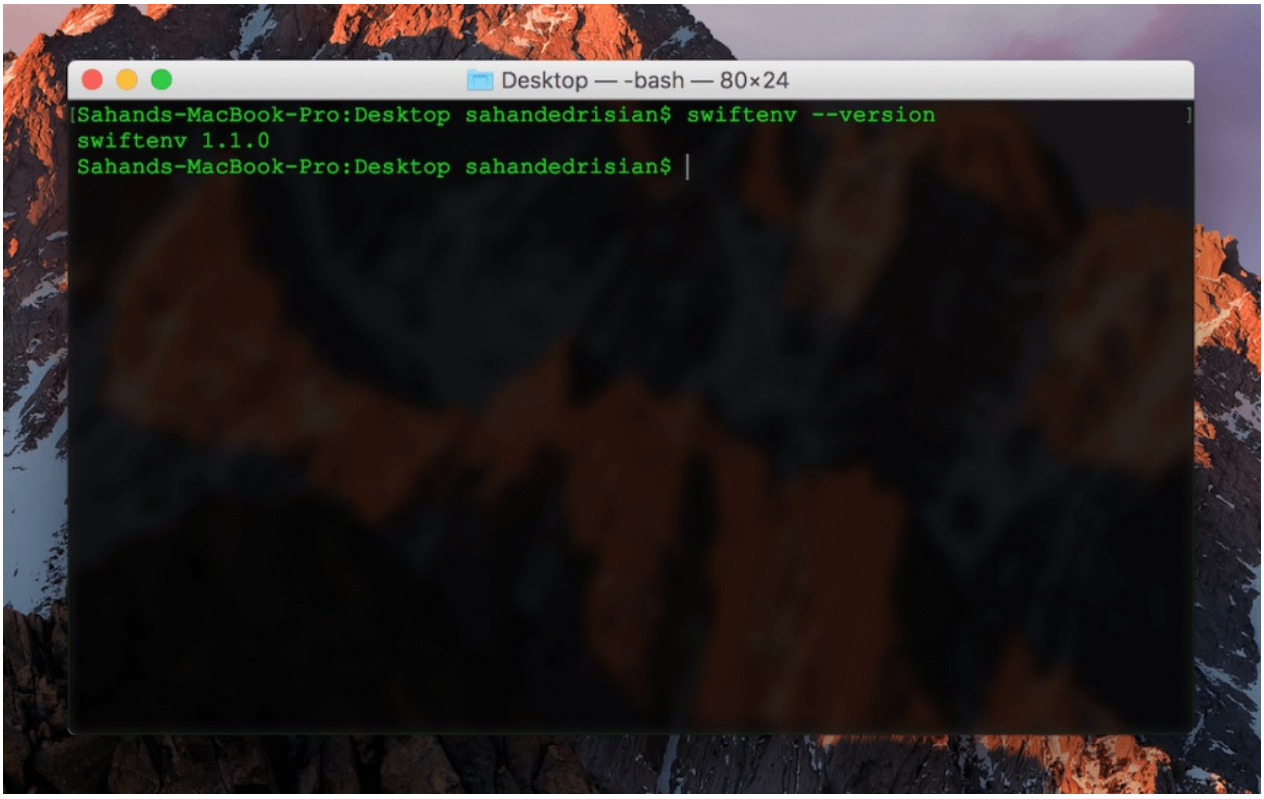
你的bash文件(在TextEdit中打开)中现在应该包含了这些命令。注意有些情况下bash文件可能包含其他命令。不要删掉之前的命令，在文件末端添加新的即可。



保存bash文件并重启终端。终端重启后，输入下面这个命令确保Swiftenv已经成功安装了:

```
swiftenv --version
```

在终端中应如下所示:



下载Swift 3快照

安好Swiftenv了，接着来下载最新的Swift 3快照(Snapshot)，Swift快照就是个打包好用于安装的Swift。执行如下命令:

```
swiftenv install 3.0-GM-CANDIDATE
```

> 更新: 若3.0-GM-CANDIDATE不好使，请使用DEVELOPMENT-SNAPSHOT-2016-09-06-a。

会花点时间下载，下载完成后在终端里将快照版本设置为主Swift版本，这将会是全局的版本，执行如下命令:

```
swiftenv global 3.0-GM-CANDIDATE
```

检查下版本是否是3.0-GM-CANDIDATE:

```
swiftenv versions
```

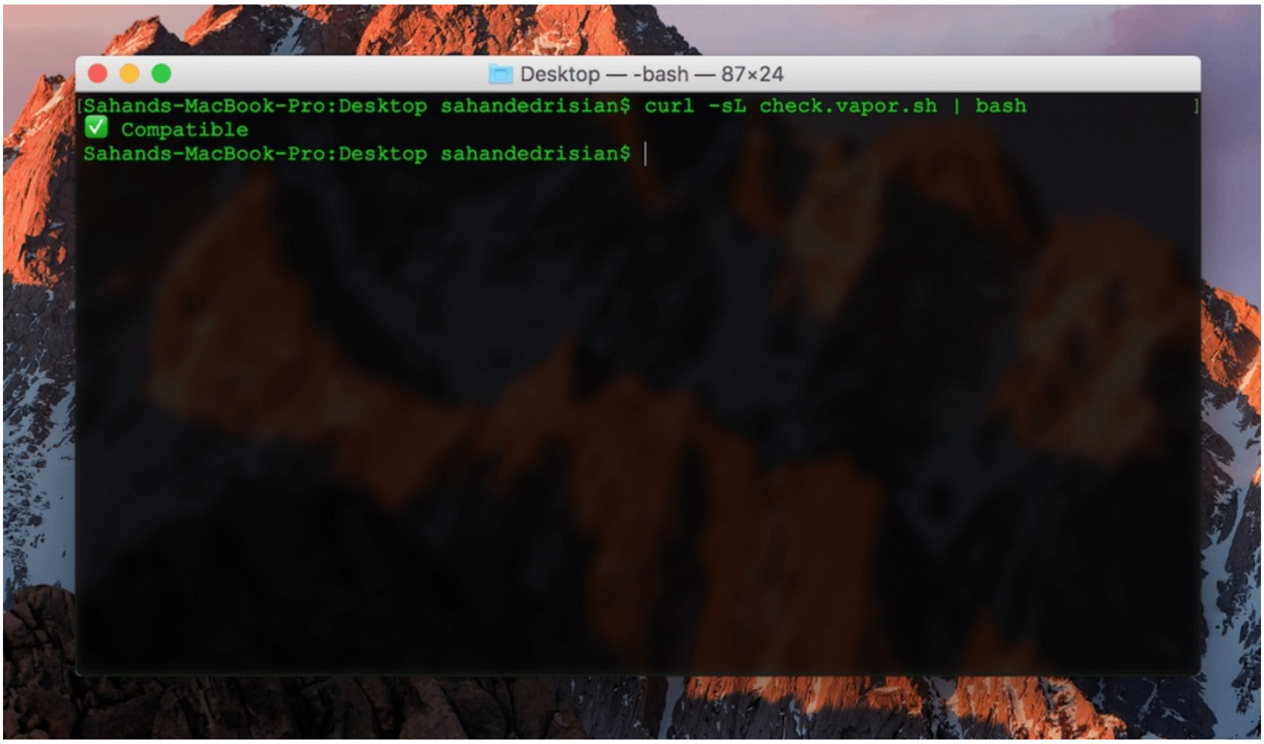
应该会看到3.0-GM-CANDIDATE后面有个星号。

完整性检验

Vapor需要合适的Swift版本才能正常启动，Vapor提供了一个shell脚本来用于检验完整性，执行如下命令来确保万事俱备:

```
curl -sL check.vapor.sh | bash
```

若显示出绿色箭头的emoji表情，就表示准备好安装Vapor了!

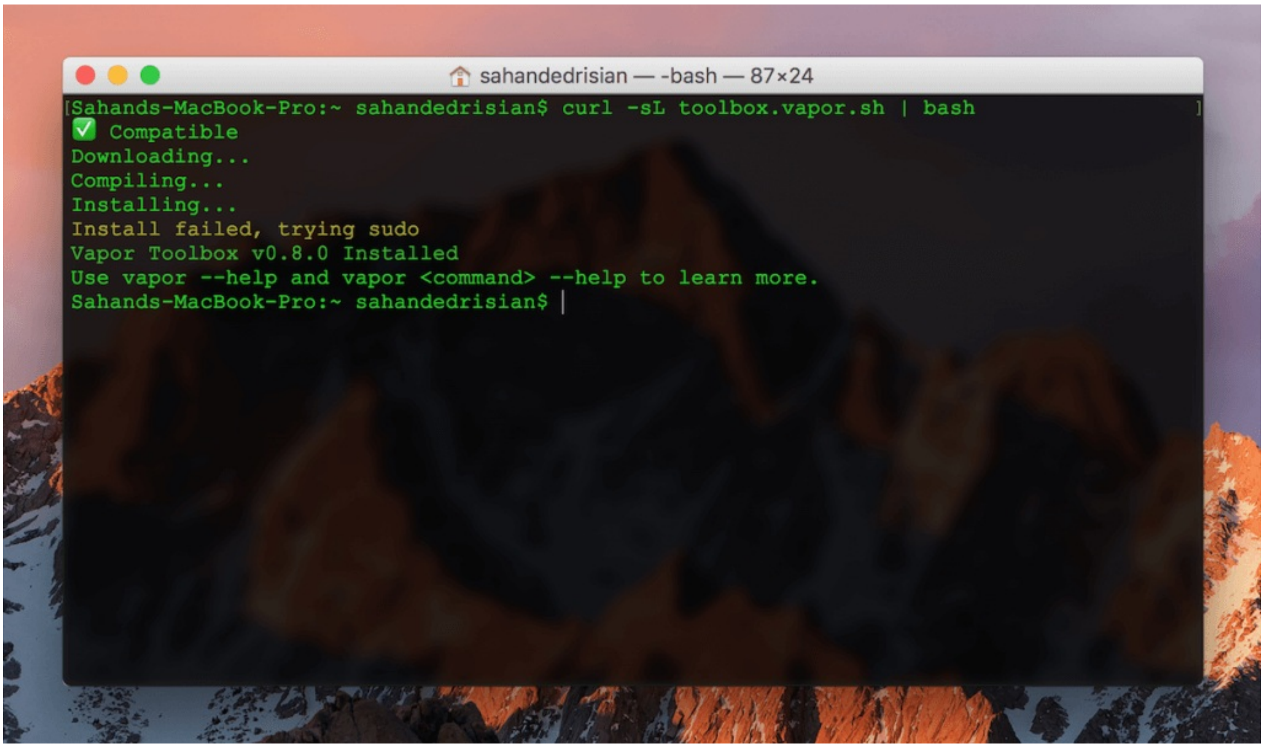


安装Vapor Toolbox

使用如下命令下载并安装Vapor:

```
curl -sL toolbox.vapor.sh | bash
```

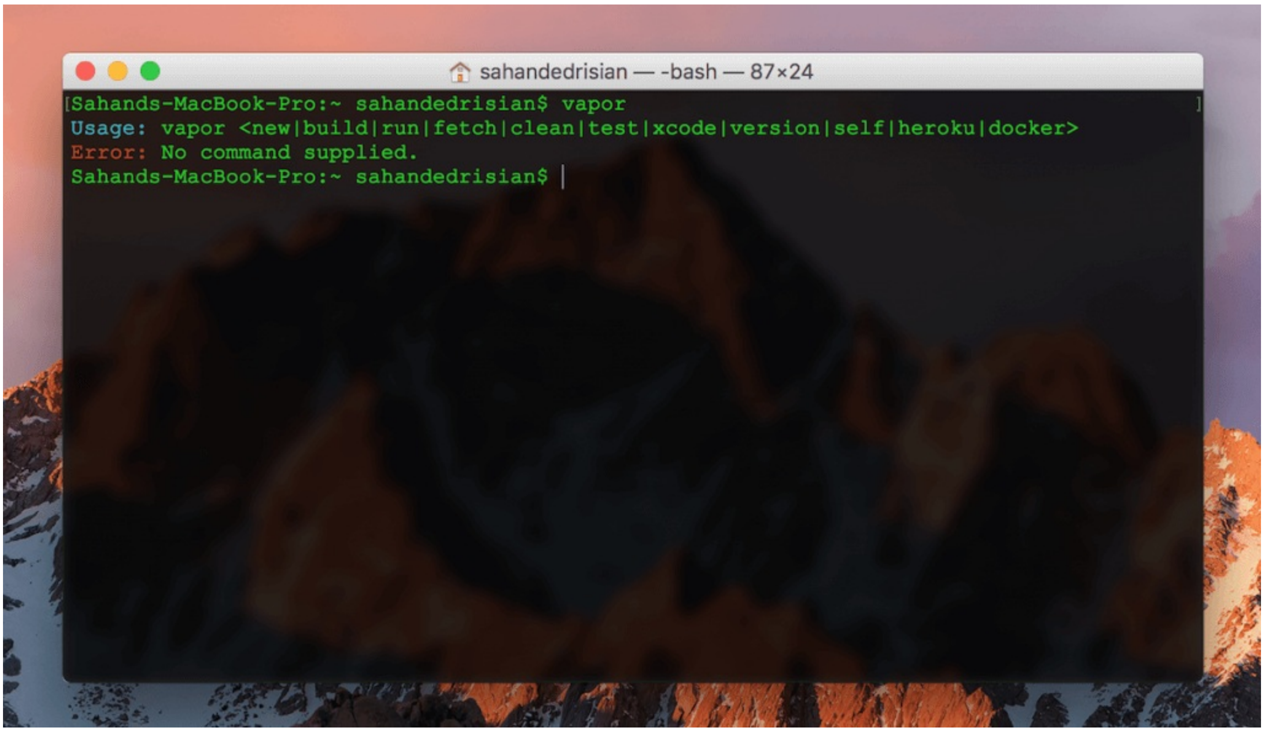
好吧，这会花点时间，也许会出错，比如"Install failed, trying sudo"。当看到"Vapor Toolbox v0.10.4 Installed"信息时就说明安装成功了。



在终端里试试如下命令，来验证下Vapor是否准备就绪：

```
vapor
```

会看到如下这样的一个命令列表：



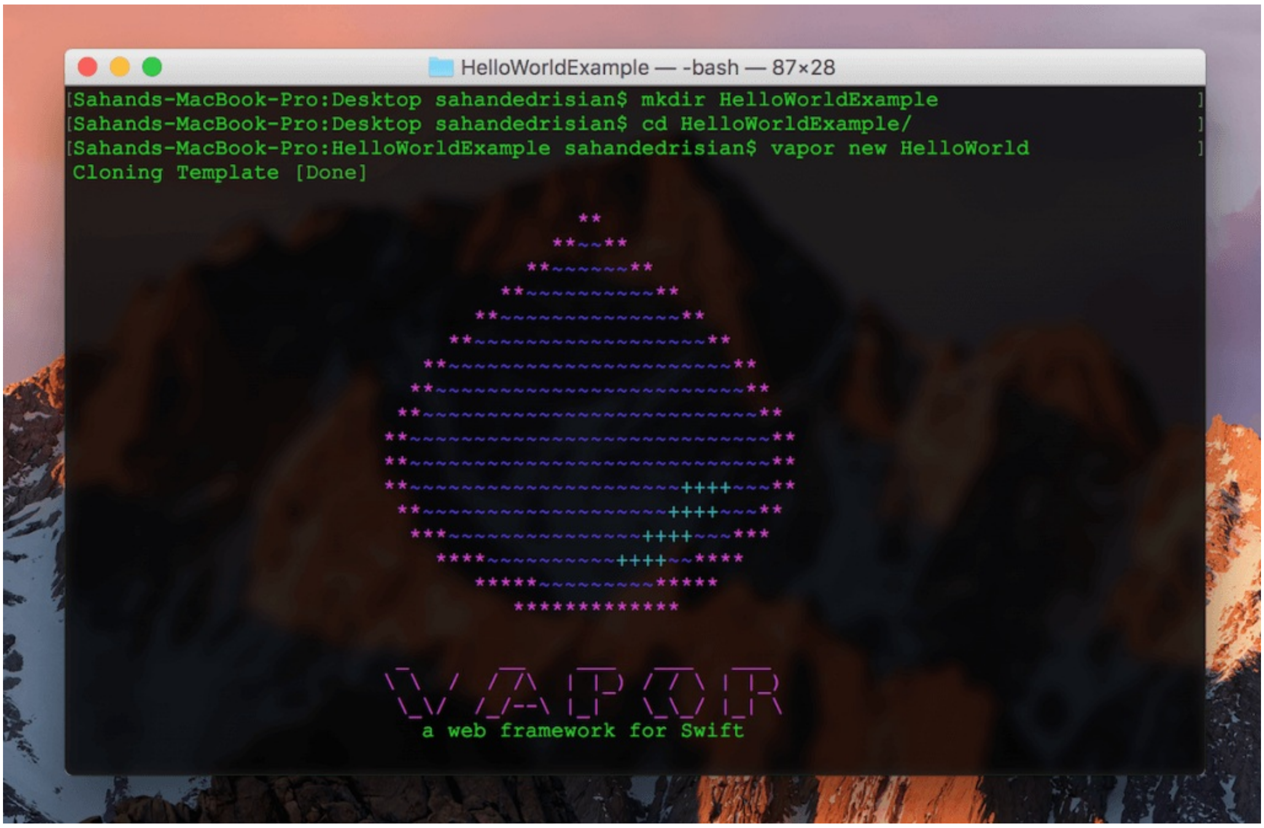
恭喜！你完成了教程中最具挑战的部分。

开始一个新的Vapor工程

所有Swift项目都使用了Swift包管理器(SPM)，每个SPM项目都需要一个Package.json文件与一个main.swift文件等等。SPM会查找项目中的Package.swift文件并下载必要的依赖库，SPM有点像NPM(Node包管理器)。Vapor会创建Swift工程并将所需的依赖添加到Package.swift文件中。

在创建工程时需要给Vapor提供一个项目名称，首先进入一个想要保存新工程的目录下，执行如下命令:

```
vapor new HelloWorld
```



Vapor会在当前的目录下创建一个名为HelloWorld的工程，使用如下命令进入新工程的目录下，目录名称与工程名称是相同的:

```
cd HelloWorld
```

会看到一个如下的目录:

```
├── Package.swift
├── App
|    ├── main.swift
|    ├── Controllers
|    ├── Middleware
|    └── Models
├── Resources
 |      └── Views
├── Config
├── Localization
└── Public
```

Vapor是遵循MVC(模型-视图-控制器)模式的框架，因此它会创建对应的文件夹来存放模型，视图与控制器，若你不太熟悉MVC设计模式，点击这里学习一下。

首先，Vapor创建好了Package.swift文件，不需要管它。然后是App文件夹，它包含了模型、控制器、中间件与main.swift文件，main.swift文件是我们app的main文件，在这里会初始化服务端，Vapor会首先运行这个文件。

Resources文件夹包含Views，用来存放HTML文件与模板。Public文件夹中存放image与style文件。其他部分现在还用不上暂且不用管。

Droplet

Vapor会自动创建一个示例工程，包含控制器，模型，中间件与main.swift文件。打开main.swift文件(在/Sources/App目录下)，清空里面的代码。

Droplet是Vapor服务器的心脏，它包含了大量后端服务器所使用的函数。首先，在main.swift文件中导入Vapor:

```
import Vapor
```

接着来创建一个Droplet:

```
let drop = Droplet()
```

Droplet中有很多自定义的属性，初始化时可接受多个命令，在Vapor的文档中都可以查到。目前阶段我们不需要自定义Droplet。

现在来处理一下对网页的'/'或index请求:

```
drop.get("/") { request in
    return "Hello World!"
}
```

最后，需要调用serve()函数来启动服务器。

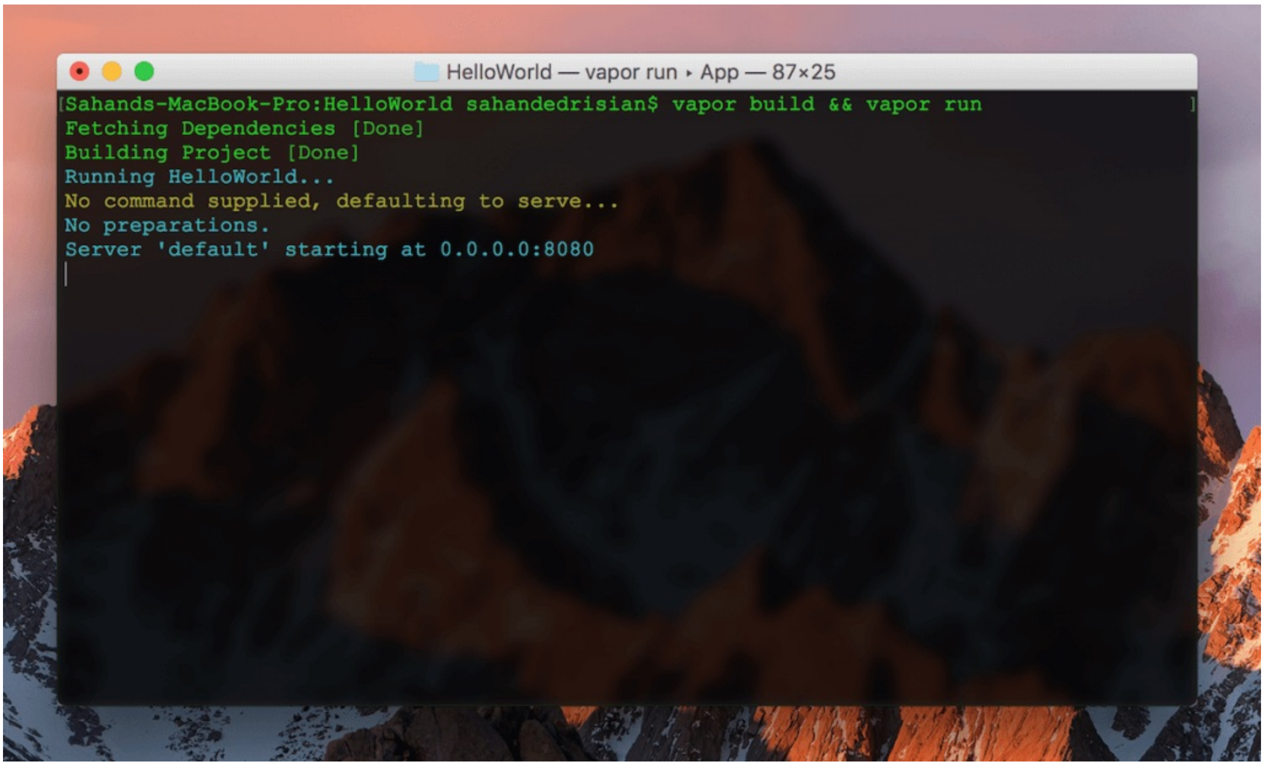
```
drop.run()
```

保存main.swift文件，现在Vapor可以构建并运行服务器了，执行如下命令:

```
vapor build
vapor run
```

Vapor没有"特殊"的构建或执行参数，简单的启动 `vapor build` 命令即可，输入 `vapor run` 命令会在 `.build/debug/` 目录下进行构建。

这个操作第一次进行时可能会花点时间，构建执行成功后会看到如下界面:



Vapor一开始会启动服务器的8080端口，若你想修改端口号，必须在servers.json文件中修改设置:

```
├── Package.swift
├── App
├── Resources
├── Config
  |     └── servers.json  <--
├── Localization
└── Public
```

假定你没有修改8080端口，可以访问下面的链接来访问服务器:

```
0.0.0.0:8080
```

或

```
localhost:8080
```

应该会看到 “Hello World” 字样!

> 注意: 每次改动后都需要build与run一下，如果仅仅执行了run命令则会执行之前构建好的工程。

处理HTTP请求

处理HTTP请求的方法跟其他比如Express，Flask等框架的做法挺像的，这篇教程中会演示GET方法的处理。

假设，现在想要在打开"localhost:8080/name/John"地址时输出字符串"Hello John!":

```
drop.get("/name",":name") { request in
    if let name = request.parameters["name"]?.string {
        return "Hello \(name)!"
    }
    return "Error retrieving parameters."
}
```

Droplet中用于处理GET的函数get()，get()函数可以接受多个参数，第一个参数是GET请求的参数名称，后面的参数是请求中参数的key值，比如:

```
drop.get("route", ":key", "route2", ":key2")
```

使用这种方法可以通过提供字典中的key值来访问参数中对应的具体指。输入key时必须在前面加上冒号，表示这是个参数的key值。

get()函数提供了一个request对象，这个对象包含与GET请求相关的所有信息。使用request对象的parameters字典来访问提交的参数:

```
request.parameters["key"]
```

使用if let语句来确保参数中的值存在:

```
if let name = request.parameters["name"].string {
    // Do something
}
```

HTTP请求需要一个来自服务端的响应。在这里，我们返回来自参数字典的 `name` 字符串。

构建并运行工程，使用如下地址来测试一个GET请求:

```
http://localhost:8080/name/John
```

应该会显示 “Hello John!”

注意，这里的路由规则与其它框架的规则(比如Express)是不同的，在Vapor中，你不能使用URI格式来访问参数:

```
http://localhost:8080/name?name=John&age=18
```

每个参数前都必须使用斜杠:

```
http://localhost:8080/name/John/age/18
```

不过若你想用URI格式的话，可以使用request对象的uri属性。想了解更多请看request对象的文档。

返回视图

Droplet可以给客户端返回一个视图，视图就是保存在/Resources/Views文件夹下的HTML文件。从这里下载一个示例view.html文件，这是个很简单的渲染网页的HTML文件，下载后将它移动到/HelloWorld/Resources/Views中:

```
├── Package.swift
├── App
├── Resources
 |    └── Views     <--
├── Config
├── Localization
└── Public
```

上面这个目录就是Droplet查询视图文件的地方，现在来实现具体代码吧，打开main.swift文件，处理一下/view路径:

```
drop.get("/view") { request in
    return try drop.view("view.html")
}
```

Droplet中有一个接受字符串参数的view()函数，参数就是HTML文件的文件名。

view()函数可能会抛出一个异常，因此使用try关键字标记是很必要的。若视图不存在，则函数会抛出一个异常，可使用catch关键字来捕捉异常，不过在这里没有必要这样因为我们知道文件是存在的。

构建并运行一下:

```
vapor build
vapor run
```

访问这个地址:

```
http://localhost:8080/view
```

应该会看到包含“Hello World”字样的视图。访问Views的文档来学习更多有关模板与渲染的内容。

部署到Heorku

首先，创建一个Github仓库，我给仓库起名为VaporExample，到Vapor工程的目录下初始化git:

```
git init
```

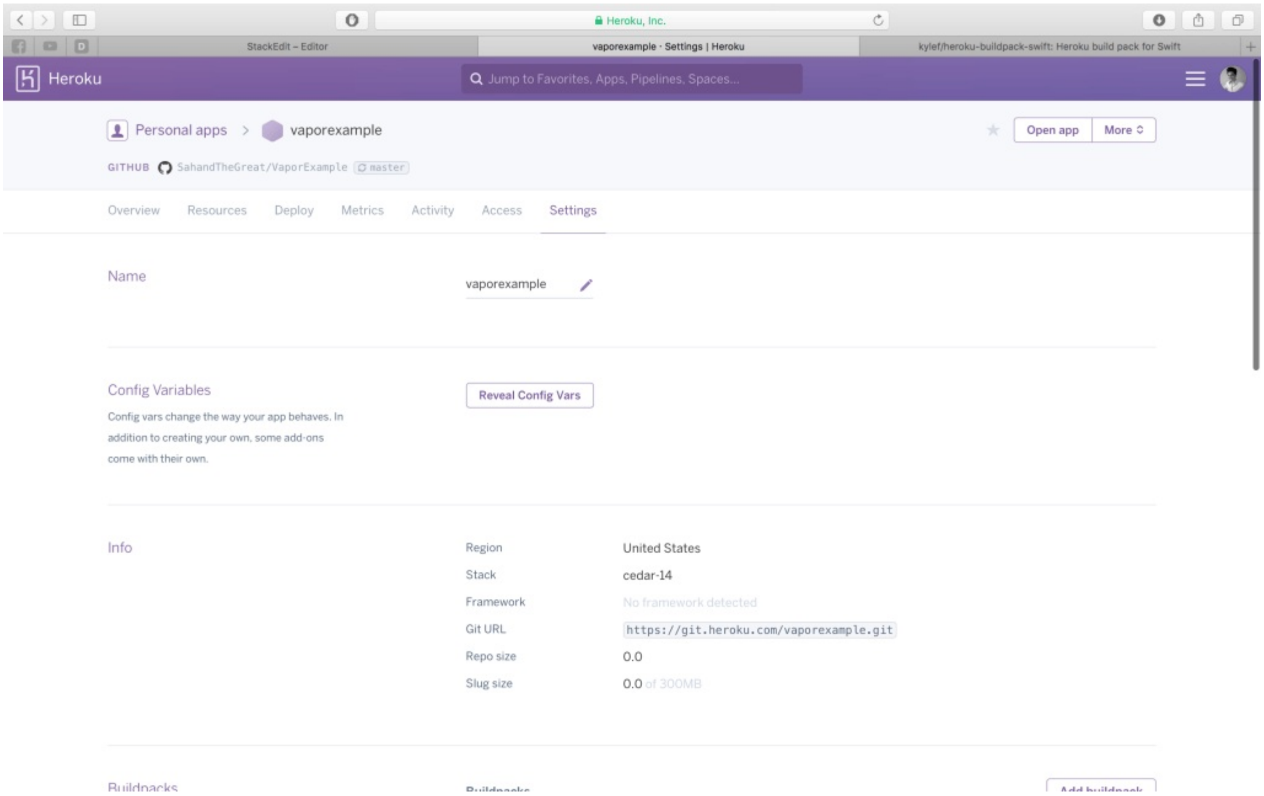
接着设置remote origin:

```
git remote set origin < 这里是你的Github仓库 >
```

添加文件, 提交修改, 然后将仓库push到Github:

```
git add .
git commit -m "Init"
git push -u origin master
```

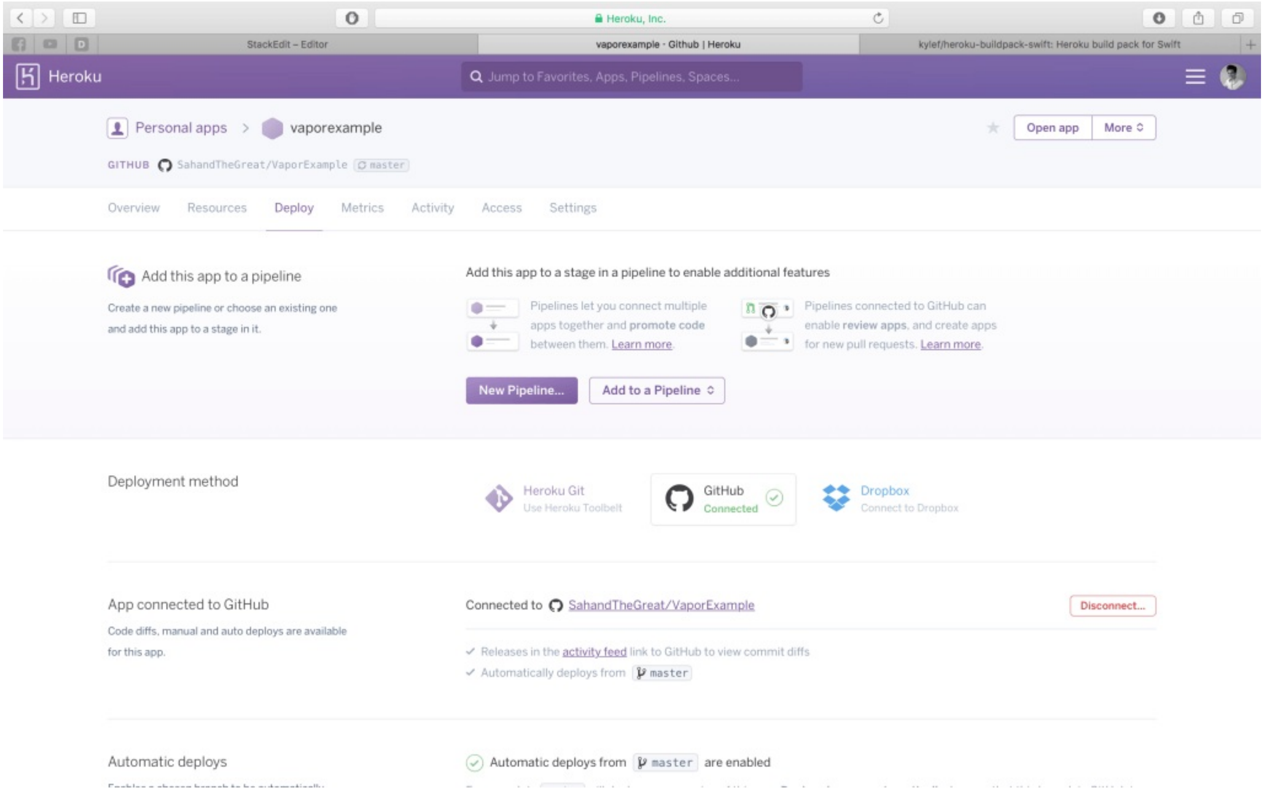
哦了，现在完成将仓库push到Github上的工作了。前往Heroku控制台，创建一个新的app，随意起名。首先来到settings选项卡中。



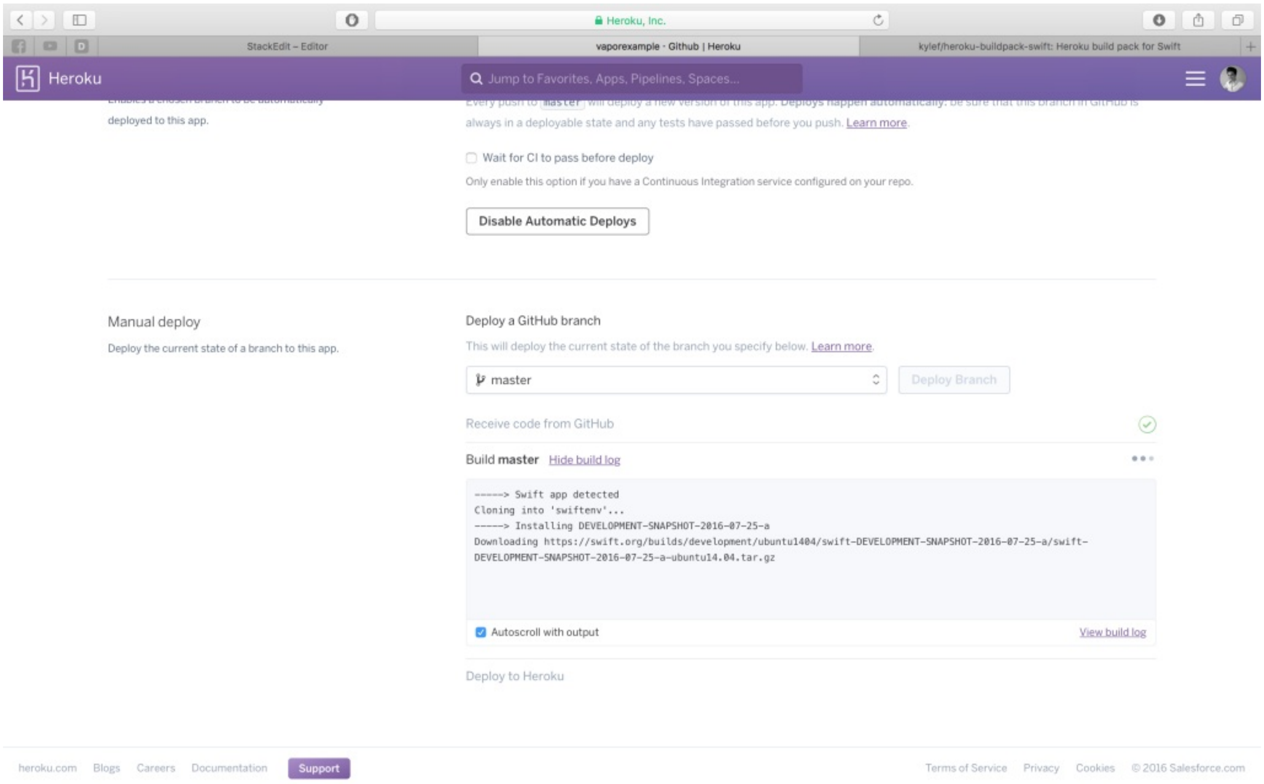
接着，向下滚动到“Buildpacks”选项。

添加一个Buildpack: https://github.com/kylef/heroku-buildpack-swift 。Buildpack构建你的Swift工程并且告诉Heroku工程所使用的语言。

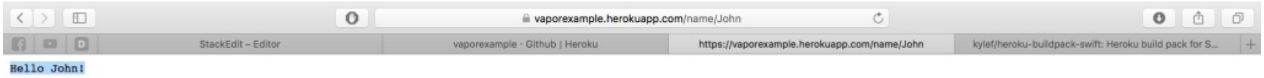
下面前往Deploy区域。



将deployment method改为GitHub，并添加Github的仓库名，将你的仓库与Heroku连接起来。向下滚动到Manual Deploy部分，点击Deploy Branch，构建过程可能会花点时间。



构建完成后，访问Heroku提供的地址来查看你的站点!



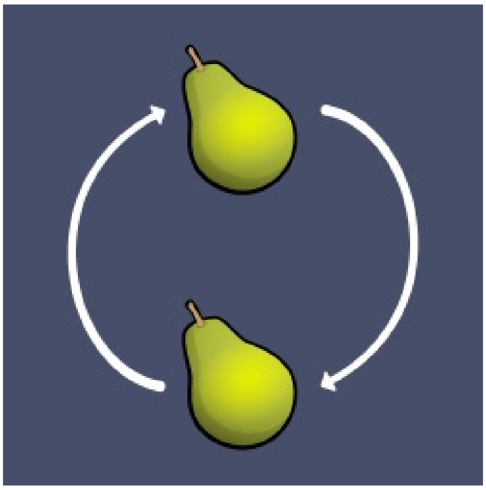
总结

希望你可以享受学习Vapor的过程！可以在Github上下载到完整工程，同时这里也有一些Vapor作者所写的示例项目。

ARC and Memory Management in Swift

Swift中的ARC与内存管理

> - 原文链接：ARC and Memory Management in Swift
> - 原文作者：Maxime Defauw
> - 译者：yrq110



Swift作为一个现代的高级编程语言，为app处理了大量内存管理的任务，协助分配与释放内存。她使用一个叫做ARC(自动引用计数,Automatic Reference Counting)的技术来完成这个工作。在这篇教程中，你会学到所有与Swift中ARC和内存管理有关的内容。

理解了ARC的原理后可以掌控堆对象的生命周期。使用ARC的Swift在资源受限的环境中(比如iOS)具有很强的预见性与有效性。

ARC是"自动"的，因此你不需要直接参与对象引用的计数过程，不过为了避免内存泄漏你需要考虑对象间的相互关系，对于新手开发者来说这个是个经常被忽略的重要问题。

在教程中会学习以下内容:

- ARC是如何工作的
- 循环引用是何物与如何中断它们
- 一个循环引用的实例，如何使用新版Xcode的可视化工具来检测
- 如何处理由值类型与引用类型混合构成的循环引用

入门

打开Xcode新建一个playground，点击File\New\Playground...，选择iOS平台，命名为MemoryManagement，保存在你想要的地方，删掉模板代码后保存一下。

接着在playground中添加如下代码:

```
class User {
  var name: String

  init(name: String) {
    self.name = name
    print("User \(name) is initialized")
  }

  deinit {
    print("User \(name) is being deallocated")
  }
}

let user1 = User(name: "John")
```

这里定义了一个User类并且创建了一个实例，User类具有name属性、init方法(在分配内存后调用)和deinit方法(释放前调用)。print语句用来帮助你了解发生了什么。

你会发现在playground的侧边栏会显示"User John is initialized\n"，对应init方法中的print语句。不过，你也会发现deinit方法中的print语句没被调用过。这意味着对象没有被反初始化，即未被释放。这是因为它所初始化的区域一直未关闭，并且playground自身未超出这个域，因此对象没有从内存中移除。

将user1的初始化语句放到do的代码块中:

```
do {
  let user1 = User(name: "John")
}
```

这里创建一个包含user1对象初始化部分的域，在域的最后我们希望user1被释放掉。

现在就能在侧边栏看到初始化和反初始化所对应的print语句输出了，这表示对象在从内存中移除之前，在域的最后被反初始化了。

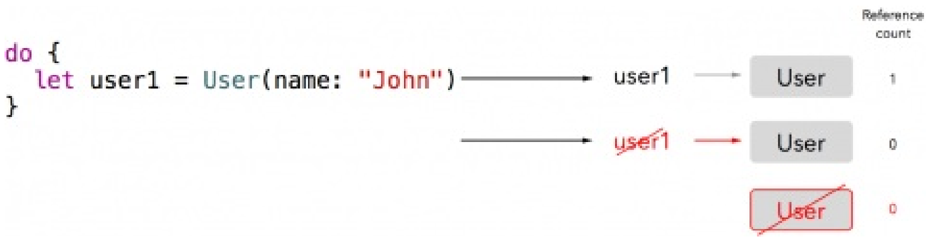
Swift对象的生命周期由5个阶段组成:

1. 分配内存 (从堆栈中取得内存空间)
2. 初始化 (运行初始化代码)
3. 使用 (对象被使用)
4. 反初始化 (运行反初始化代码)
5. 释放内存 (把内存返回给堆栈)

在分配和释放过程中没有钩子的情况下，可以在初始化和反初始化时使用print语句作为一个代理来监视进程行为。有时“deallocate(释放)”与“deinit(反初始化)”会交替使用，不过它们实际上是一个对象生命周期中两个不同的阶段。

引用计数(reference counting)根据的是当一个对象不再被需要时会被释放的原理，那么这里有个问题“你怎么能确定一个对象在未来不被需要?”，引用计数中通过使用一个count(计数)来管理，就是所说的引用计数，存在于每个对象实例中。

这个计数表示有多少“东西”引用了对象。当一个对象的引用计数变为0时，没有一个“用户”保留对象，则这个对象会被反初始化和释放。



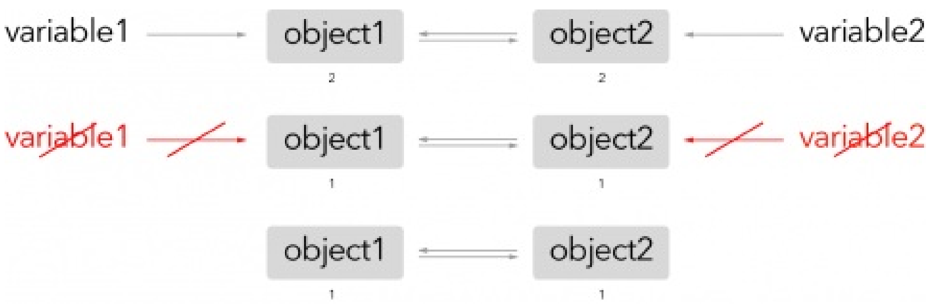
当初始化User对象时，在user1引用该对象后它的引用计数会从1开始计算。在do块的最后，由于use1超出了这个域，则计数减一，引用计数减到了0，那么use1会被反初始化，紧接着被释放。

循环引用

多数情况下ARC是很有效的，作为一个开发者，无需总是担心内存泄漏的问题——无用对象一直占用着内存的现象。

不过这并不会一帆风顺，可能会发生泄露的！

那么泄露是如何发生的呢? 想象一下这样一个情况：有两个不再需要的对象，其中每个都引用了另一个，由于两者都有一个非零的引用计数，那么这两个对象永远不会被释放。



这被称作循环强引用，它骗过了ARC并且拒绝被清理。如你所见，在结尾时的引用计数并不是零，因此object1与object2将永远不会被释放，即使已经不需要它们了。

来实际操作一下看看，在User类之后，do代码块之前添加如下代码:

```
class Phone {
  let model: String
  var owner: User?

  init(model: String) {
    self.model = model
    print("Phone \(model) is initialized")
  }

  deinit {
    print("Phone \(model) is being deallocated")
  }
}
```

然后把do中的语句改成下面这样:

```
do {
  let user1 = User(name: "John")
  let iPhone = Phone(model: "iPhone 6s Plus")
}
```

这里添加了一个新类Phone并创建了这个新类的实例。

这个新类很简单：包含两个属性：model和owner，两个方法：init和deinit方法。owner属性是可选的，因为一个Phone是可以脱离User存在的。

接下来，在User类中添加如下代码:

```
private(set) var phones: [Phone] = []
func add(phone: Phone) {
  phones.append(phone)
  phone.owner = self
}
```

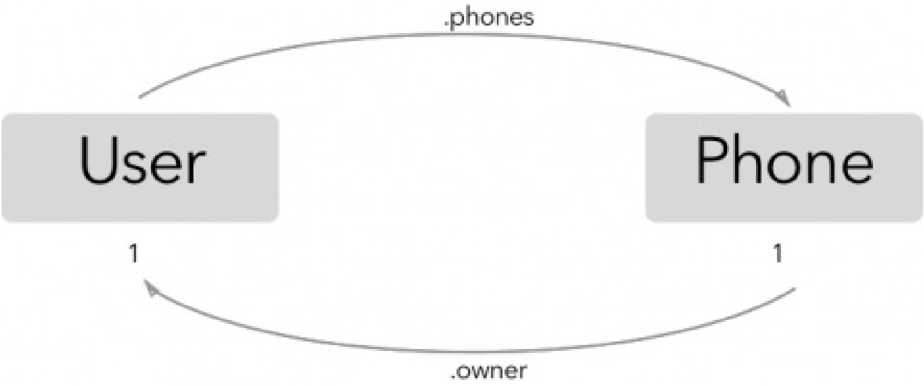
这里添加了一个phone数组来保存一个用户拥有的全部phone，将setter设为private，强制用户使用add(phone:)方法来添加phone而不是操作这个数组。这个方法确保当添加phone时有正确的用户。

现在在侧边栏中可以看到，Phone和User对象都如愿被释放了。

把do代码块改成下面这样:

```
do {
  let user1 = User(name: "John")
  let iPhone = Phone(model: "iPhone 6s Plus")
  user1.add(phone: iPhone)
}
```

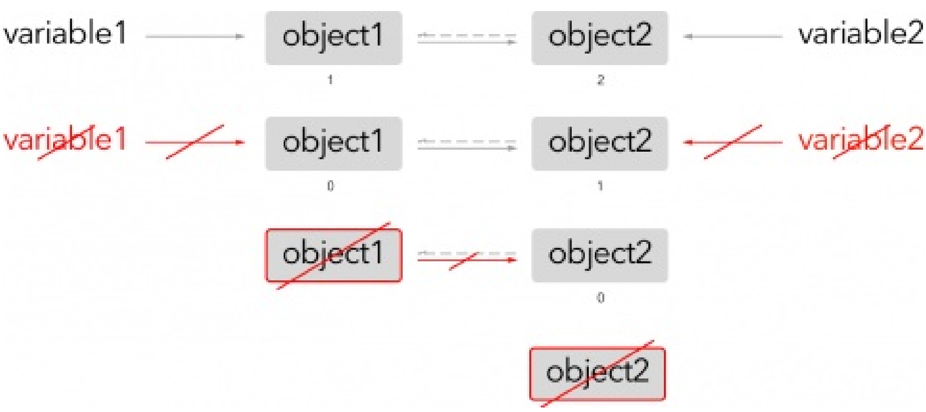
这里将iPhone添加到user1中，自动将iPhone的owner属性设为user1。两个对象间的循环强引用会欺骗ARC使其不会被释放，如此一来，use1和iPhone将永远不会释放。



弱引用

为了消除循环强引用，你可以将对象间的关系指定为弱引用，若不指明的话所有引用都是抢引用，与强引用相比，弱引用不会增加对象的强引用计数。

换句话说，弱引用不会参与对象的生命周期管理，并且弱引用是被声明为可选类型来使用的，这就意味着当引用计数变为0时，引用会自动设置为空。

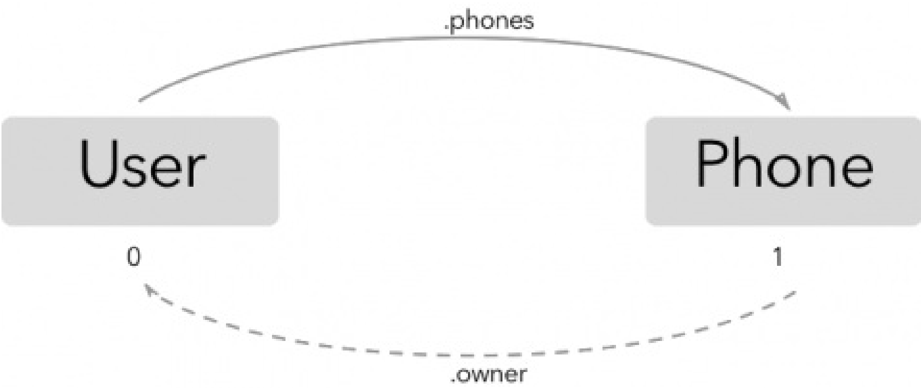


在上图中，虚线箭头表示弱引用，注意object1的引用计数为1是因为variable1引用了它。object2的引用计数是2，是因为variable2与object1都引用了它。在object2引用object1时使用的是弱引用，这意味着不会影响object1的强引用计数。

在variable1和variable2都被移除时，object1的引用计数变为0并将调用deinit方法，这样会移除object2上的强引用，则object2接下来也会被反初始化。

回到playground中，使用如下方法弱化owner的引用来中断User–Phone之间的循环引用:

```
class Phone {
  weak var owner: User?
  // 其它代码...
}
```



可以在侧边栏看到结果，在do代码块的最后user1和iPhone被正常的释放掉了。

无主引用

这里有另一个可以使用并且不会增加引用计数的引用修饰符：unowned。

无主引用与弱引用有什么不同吗? 弱引用是可选类型，当引用对象反初始化时会变为nil，这就是为何必须将weak属性定义为可选的var类型。相比之下，无主引用并不是可选类型，若尝试去访问一个反初始化后对象的unowned属性的话会触发一个运行时错误。

	Var	Let	Optional	Non-Optional
Strong	👍	👍	👍	👍
Weak	👍	⛔	👍	⛔
Unowned	👍	👍	⛔	👍

来练习一下unowned.，在do代码块的前面添加一个新的类CarrierSubscription:

```
class CarrierSubscription {
  let name: String
  let countryCode: String
  let number: String
  let user: User

  init(name: String, countryCode: String, number: String, user: User) {
    self.name = name
    self.countryCode = countryCode
    self.number = number
    self.user = user

    print("CarrierSubscription \(name) is initialized")
  }

  deinit {
    print("CarrierSubscription \(name) is being deallocated")
  }
}
```

CarrierSubscription类有四个属性: name, countryCode, number, 和一个引用的User对象。

接着在User类中添加如下代码:

```
var subscriptions: [CarrierSubscription] = []
```

这里添加了subscriptions属性，使用一个数组来保存CarrierSubscrition对象。

在Phone类的顶部，owner属性后面添加如下代码:

```
var carrierSubscription: CarrierSubscription?

func provision(carrierSubscription: CarrierSubscription) {
  self.carrierSubscription = carrierSubscription
}

func decommission() {
  self.carrierSubscription = nil
}
```

添加了一个可选CarrierSubscription属性和两个新函数-provision与decommission，用来添加和移除phone上的运营商描述。

接在在print语句之前添加如下代码，初始化内部的CarrierSubscription:

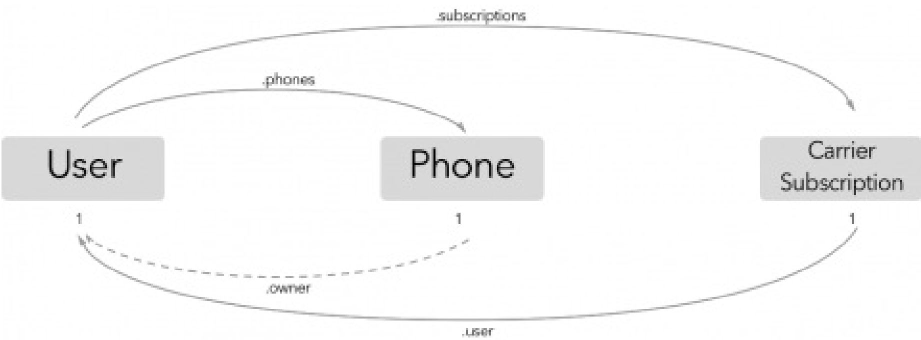
```
user.subscriptions.append(self)
```

将CarrierSubscription添加到用户的描述数组中。

最后，将do代码块中的内容改成如下所示:

```
do {
  let user1 = User(name: "John")
  let iPhone = Phone(model: "iPhone 6s Plus")
  user1.add(phone: iPhone)
  let subscription1 = CarrierSubscription(name: "TelBel", countryCode: "0032", number:
"31415926", user: user1)
  iPhone.provision(carrierSubscription: subscription1)
}
```

看看侧边栏输出了什么结果。再次看到了循环引用：不管是user、iphone还是subscription1，最后都没有被释放。现在你可以发现问题出在哪吗?



将从use1到subscription1的引用或从subscription1到user1的引用变成unowned即可中断循环，问题是选择哪一个引用？

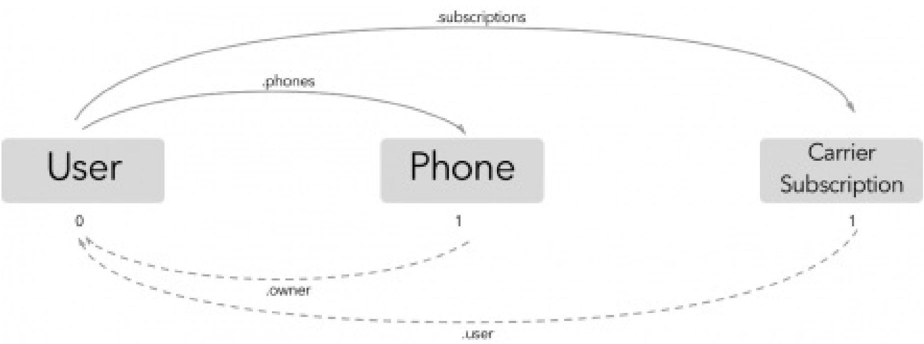
user拥有运行商描述，而运营商描述不可拥有user，并且不存在一个没有用户的CarrierSubscription，这就是为何在一开始将其声明为一个不可变的let属性。

因为可以存在一个不包含CarrierSubscription的User，不可存在一个不包含User的CarrierSubscription，所以应该用unowned来修饰user引用。

给CarrierSubscription的user属性添加unowned修饰符:

```
class CarrierSubscription {
  let name: String
  let countryCode: String
  let number: String
  unowned let user: User
  // Other code...
}
```

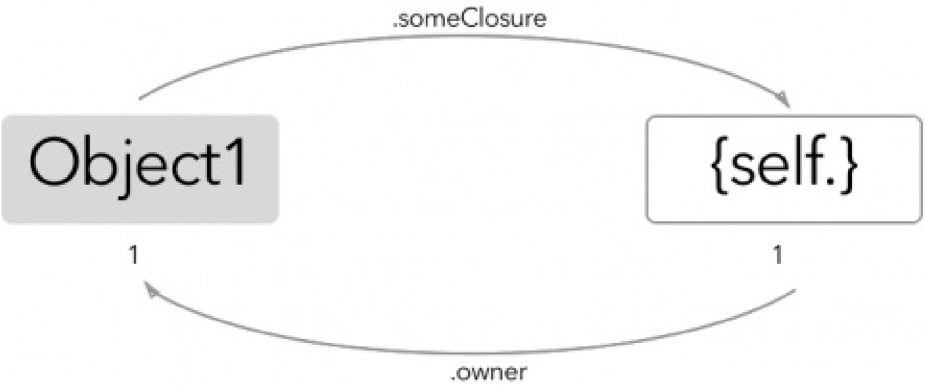
这样就中断了循环引用，使每个对象都被释放掉。



闭包的循环引用

当对象的属性相互引用时会发生循环引用。同样，闭包也是可引用的类型，也会发生循环引用的问题，闭包在这里捕捉到了所操作的对象。

举个栗子，若有一个闭包被分配给一个类的属性，并且这个闭包使用了相同类下的属性，这就发生了循环引用。换言之，对象通过一个存储属性引用了闭包，闭包通过捕捉的self值引用了对象。



在CarrierSubscription的user属性后添加如下代码:

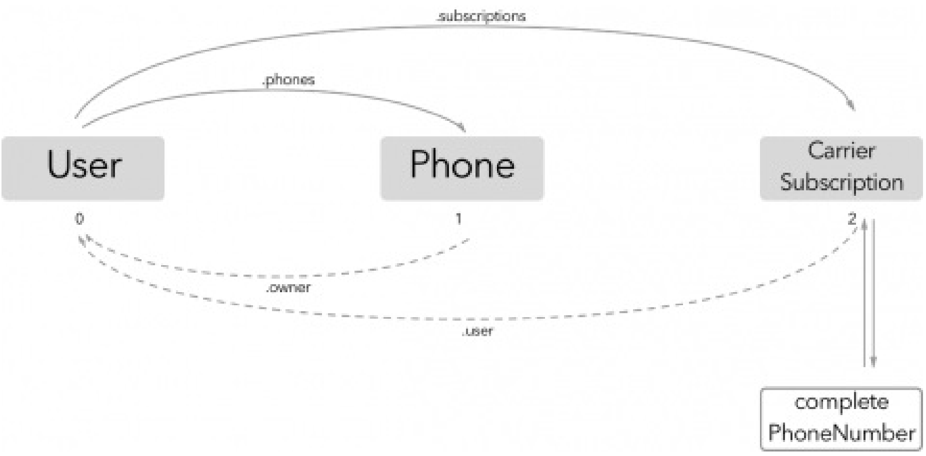
```
lazy var completePhoneNumber: () -> String = {
  self.countryCode + " " + self.number
}
```

这个闭包计算并返回一个完整的电话号码。使用lazy声明这个属性，意味着直到第一次使用时才会分配属性，使用这个修饰符是因为这个属性是依赖self.countryCode和self.number的，它们在初始化后才可用。

在do代码块的最后添加如下代码:

```
print(subscription1.completePhoneNumber())
```

你会注意到use1和iPhone都会释放了，不过因为存在的对象与闭包之间的循环强引用，CarrierSubscription并没有被释放。



Swift有一个简便、优雅的方式去中断闭包的循环强引用：在定义闭包与所捕捉的对象间的联系时声明一个捕捉列表。

为了形象的描述捕捉列表是如何工作的，试试如下代码:

```
var x = 5
var y = 5

let someClosure = { [x] in
  print("\(x), \(y)")
}

x = 6
y = 6

someClosure()        // Prints 5, 6
print("\(x), \(y)")  // Prints 6, 6
```

变量x在捕捉列表中，因此在定义闭包时会创建一个x的副本，通过它的值进行捕捉。另外，y不在捕捉列表中，通过它的引用进行捕捉。这意味着当闭包运行的时候，y会变为那个时刻的y值，而不是捕捉时的y值。

捕捉列表在使用weak或unowned定义闭包中的所用对象时会派上用场。当CarrierSubscription实例释放时闭包就不存在了，在这种情况下使用unowned比较好。

把CarrierSubscription中的completePhoneNumber闭包改成如下这样:

```
lazy var completePhoneNumber: () -> String = {
  [unowned self] in
  return self.countryCode + " " + self.number
}
```

在闭包的捕捉列表中加入了[unowned self]，这意味着self对象会作为一个unowned引用被捕捉，而不是一个强引用类型。

这样就解决循环引用的问题了，万岁!

实际上这里的语句是捕捉语句的精简版，看看加长版的格式:

```
var closure = {
  [unowned newID = self] in
  // Use unowned newID here...
}
```

这里的newID是一个self的unowned副本。在闭包域外面的self是对象本身的含义，而在上面使用的精简版格式中则创建了一个新的self变量替代了已存在的self变量。

在代码中，self与completePhoneNumber闭包之间的关联是unowned类型的。若你能确保闭包中引用的对象永远不会被释放，可以使用unowned，如果它会被释放，那你就摊上事儿了。

在playground的最后添加如下代码:

```
// A class that generates WWDC Hello greetings.  See http://wwdcwall.com
class WWDCGreeting {
  let who: String

  init(who: String) {
    self.who = who
  }

  lazy var greetingMaker: () -> String = {
    [unowned self] in
    return "Hello \(self.who)."
  }
}

let greetingMaker: () -> String

do {
  let mermaid = WWDCGreeting(who: "caffinated mermaid")
  greetingMaker = mermaid.greetingMaker
}

greetingMaker() // TRAP!
```

playground会引起一个运行时异常，因为闭包期望self.who是可用的属性，不过当美人鱼(mermaid)游到域外它就被释放了。虽然这个例子有点牵强，不过在真实环境中是很容易发生的，比如使用闭包运行一些域外的事物，或者在一个异步网络调用完成之后。

把WWDCGreeting中的greetingMaker变量改成如下所示:

```
lazy var greetingMaker: () -> String = {
  [weak self] in
  return "Hello \(self?.who)."
}
```

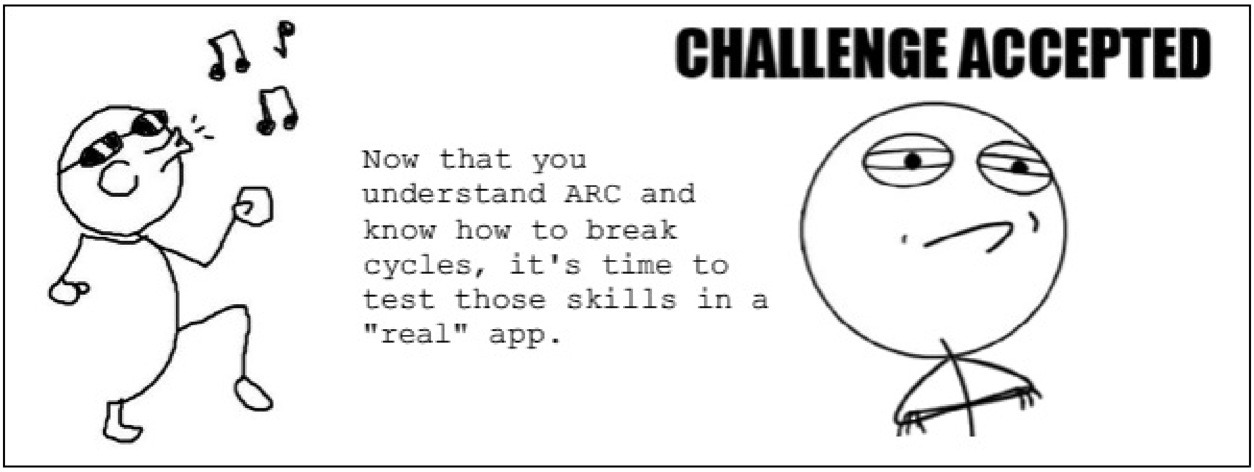
这里对原本的greetingMaker闭包做了两项改动，首先使用weak替换了unowned，其次，由于self变成了weak，因此需要使用self?.who来访问who属性。

这样playground就不会崩溃了，不过在侧边栏得到了一个令人好奇的结果: "Hello, nil."也许这个也可以接受，不过若你想在对象释放时搞一些完全不同的事情的话，Swift的guard let修饰符可以帮助你。

最后一次修改这个闭包，如下所示:

```
lazy var greetingMaker: () -> String = {
  [weak self] in
  guard let strongSelf = self else {
    return "No greeting available."
  }
  return "Hello \(strongSelf.who)."
}
```

guard语句绑定了一个新的strongSelf变量。若self为空，则闭包返回 "No greeting available."。另一方面，若self非空，则strongSelf创建一个强引用，这样对象就会被保留直到闭包结束。

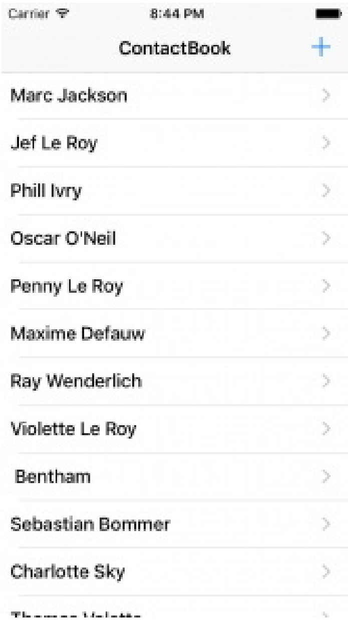


在Xcode 8中发现循环引用

现在你已经理解了ARC的原理，了解了循环引用与中断它们的方法，是时候用一个真实的例子上手了。

下载这个开始工程 并用Xcode 8打开它。需要Xcode 8是因为要使用一些Xcode 8中新增的有趣特性。

构建并运行工程，会看到这样一个界面:



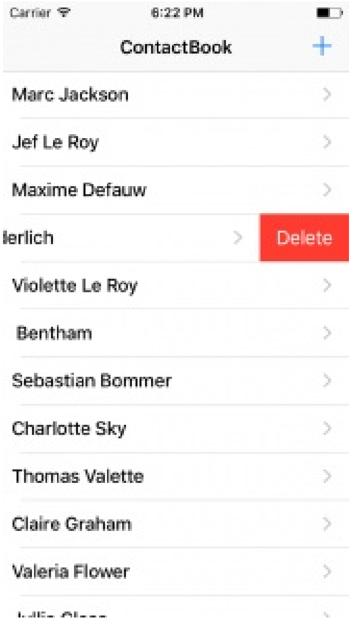
这是一个简单的通讯录app，试着随意点击一个联系人获取更多信息或者点击右上角的加号添加联系人。

梳理一下代码:

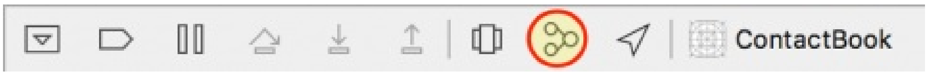
- ContactsTableViewController: 显示数据库中的所有Contact对象。
- DetailViewController: 显示一个选择Contact对象的详细信息。
- NewContactViewController<: 允许用户添加新联系人。
- ContactTableViewCell: 显示联系人对象详细信息的自定义tableview cell。
- Contact: 数据库中联系人的model。
- Number: 电话号码的model。

不巧的是，工程中出现了一些严重的错误：被循环引用毁了。你的用户短时间内不会发现这些问题，因为泄露的对象都非常小，并且小到难以被trace。幸运的是，Xcode 8有一个新的内置工具，可以帮助你找到这些很细微的泄露。

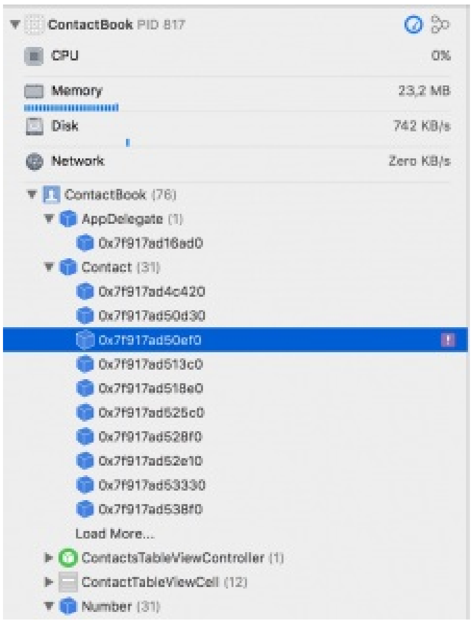
再次构建并运行工程，向左滑动点击delete删除三四个联系人，的确完全消失了，对吗?



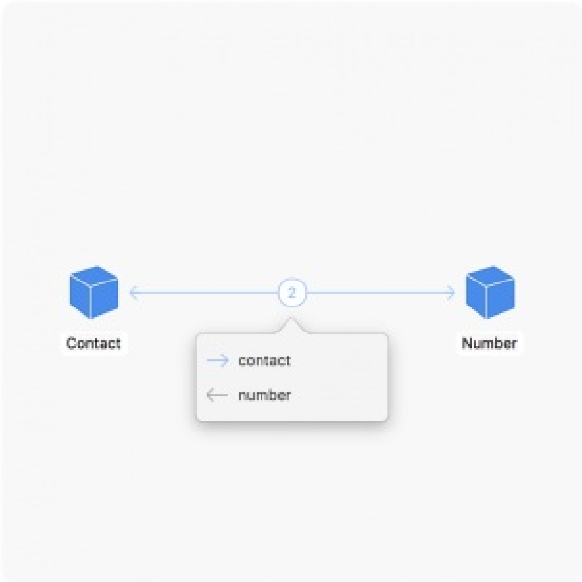
在app运行的时候到Xcode的底部，点击Debug Memory Graph按钮。



观察Xcode 8中新一类issue：Runtime Issue，图标是一个紫色方块中包含一个白色的感叹号，如在下图选择的这项这样:



在Navigator中选择一个有问题的Contact对象。可以清楚的看到循环：Contact与Number对象通过引用对方使双方一直存在。



这种添加的标记会帮助你浏览代码。想一下，一个联系人可以没有号码，不过一个号码不能没有联系人，那么如何处理这个循环？需要将从联系人到号码的引用或从号码到联系人的引用设置为weak或unowned吗？

这里有两种可能的解决方式：可以将从联系人到号码的引用设为weak型，或将从号码到联系人的引用设为unowned型，来解决这个循环。

苹果的文档中推荐一个方法：父类对象通过一个强引用的形式来保留子对象，也就是说让联系人使用强类型引用号码，号码使用无主类型引用联系人:

```
class Number {
  unowned var contact: Contact
  // Other code...
}
```

```
class Contact {
  var number: Number?
  // Other code...
}
```

再次运行并debug，发现问题解决了！

值类型与引用类型的循环

Swift中的类型可以分为引用类型(如类)与值类型(如结构体或枚举)，主要的不同在于值类型在被引用时会创建自己的副本，而引用类型则会分享一个简易的引用信息(PS：类似指针)。

那么这是不是意味着使用值类型就不会出现循环引用？是的：所有东西都是值类型的副本，不存在循环引用，这是由于没有创建任何真正的引用，毕竟至少有两个引用才能构成一个循环。

回到playground中，在最后添加如下代码:

```
struct Node { // Error
  var payload = 0
  var next: Node? = nil
}
```

哇哦，编译器有小情绪了。一个结构体(值类型)是不能迭代或使用自身实例的，否则这种结构体的尺寸就会变为无限大。把代码改成下面这样的类:

```
class Node {
  var payload = 0
  var next: Node? = nil
}
```

对于类来说自身引用不叫事儿，因此没有编译错误出现。

接着在playground中添加如下代码:

```
class Person {
  var name: String
  var friends: [Person] = []
  init(name: String) {
    self.name = name
    print("New person instance: \(name)")
  }

  deinit {
    print("Person instance \(name) is being deallocated")
  }
}

do {
  let ernie = Person(name: "Ernie")
  let bert = Person(name: "Bert")

  ernie.friends.append(bert) // Not deallocated
  bert.friends.append(ernie) // Not deallocated
}
```

这是一个由值类型与引用类型混合组成的循环引用示例。

由于ernie和bert的friends数组中相互引用了对方因此会一直存在，即使数组本身是值类型，但是把数组改为unowned的话Xcode会出错：unowned只能对类类型执行。

为了中断这里的循环，需要创建一个泛型包装对象来添加数组的实例。若不知道泛型的话，可以看看网站中的泛型指南。

在Person类定义的前面添加如下代码:

```
class Unowned<T: AnyObject> {
  unowned var value: T
  init (_ value: T) {
    self.value = value
  }
}
```

接着改下Person中friends属性的定义:

```
var friends: [Unowned<Person>] = []
```

最后，把do代码块中的代码改成下面这样:

```
do {
  let ernie = Person(name: "Ernie")
  let bert = Person(name: "Bert")

  ernie.friends.append(Unowned(bert))
  bert.friends.append(Unowned(ernie))
}
```

现在ernie和bert被成功释放掉了!

friends数组不再是一个Person对象的集合了，取而代之成为了一个由Person实例包装后的Unowned对象组成的集合。

可以像下面这样使用值属性访问Unowned对象中的Person实例:

```
let firstFriend = bert.friends.first?.value // 得到ernie
```

A Beginner’s Guide to Auto Layout with Xcode 8

Xcode 8中的Auto Layout新手指南

> - 原文链接 : A Beginner’s Guide to Auto Layout with Xcode 8
> - 原文作者 : SIMON NG
> - 译者 : yrq110

Auto layout是一个基于约束(constraint-based)的布局系统，允许开发者做出可以响应屏幕尺寸与设备方向的UI组件。一些新人开发者因为它的难度而敬而远之，有些开发者则不去用它。不过相信我，今天来用它开发一个app后你会爱不释手的。

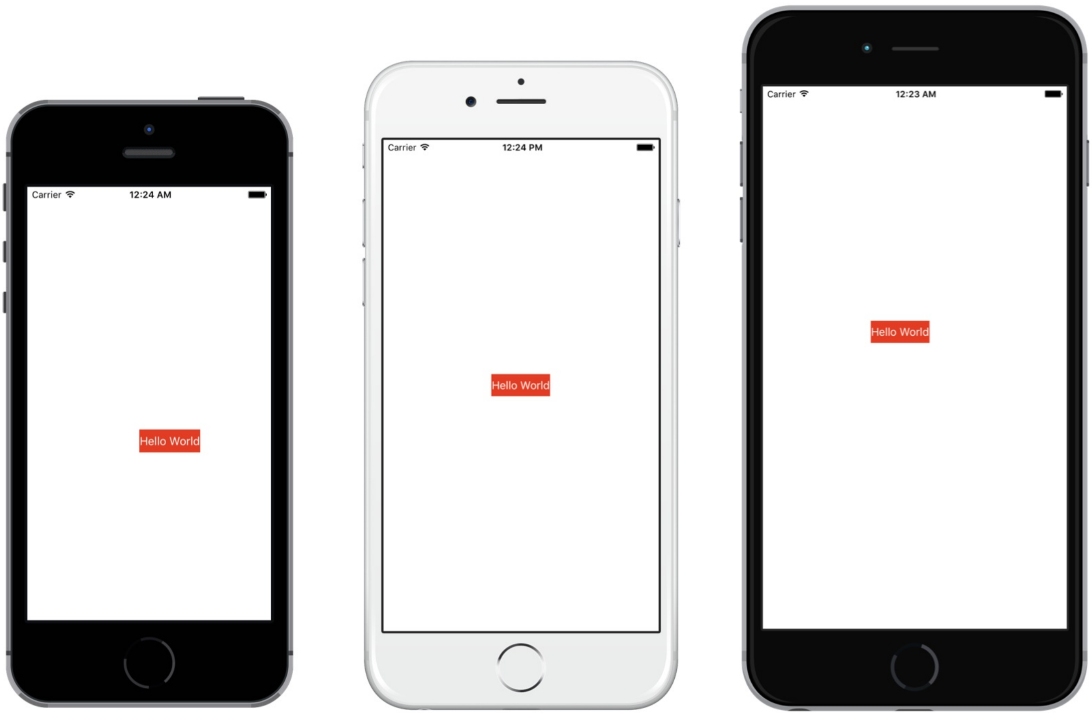
随着iPhone 6/6s、6/6s Plus的发布，苹果手机的屏幕尺寸包含了3.5英寸, 4英寸, 4.7英寸和5.5英寸这几种。若不用auto layout的话很难创建一个适配所有屏幕尺寸的app，并且从Xcode 6开始必然会使用auto layout去设计用户界面。

在这篇教程中，会帮你通过使用auto layout建立一个设计响应式UI的坚实基础。

Auto layout并不像人们想象中的那么难。理解它的基础概念与使用方法后，就可以使用auto layout为所有iOS设备做出复杂的用户界面。

为何使用Auto Layout?

来看个例子，下载并运行开始工程。不要使用iPhone 6/6s模拟器，使用iPhone 6 Plus或iPhone 5模拟器运行，会得到下图的结果，按钮在除了iPhone 6/6s外的其他iPhone设备上都不在屏幕中心。



iPhone 5s　　iPhone 6　　iPhone 6 Plus

来再试一下。

点击Stop按钮，使用iPhone 6/6s模拟器运行，模拟器启动后，在菜单中选择 Hardware > Rotate Left (or Rotate Right)，这会使设备旋转到水平模式。也可以按下command+左箭头/右箭头来旋转设备方向。这一次，Hello World按钮也没有在中间。

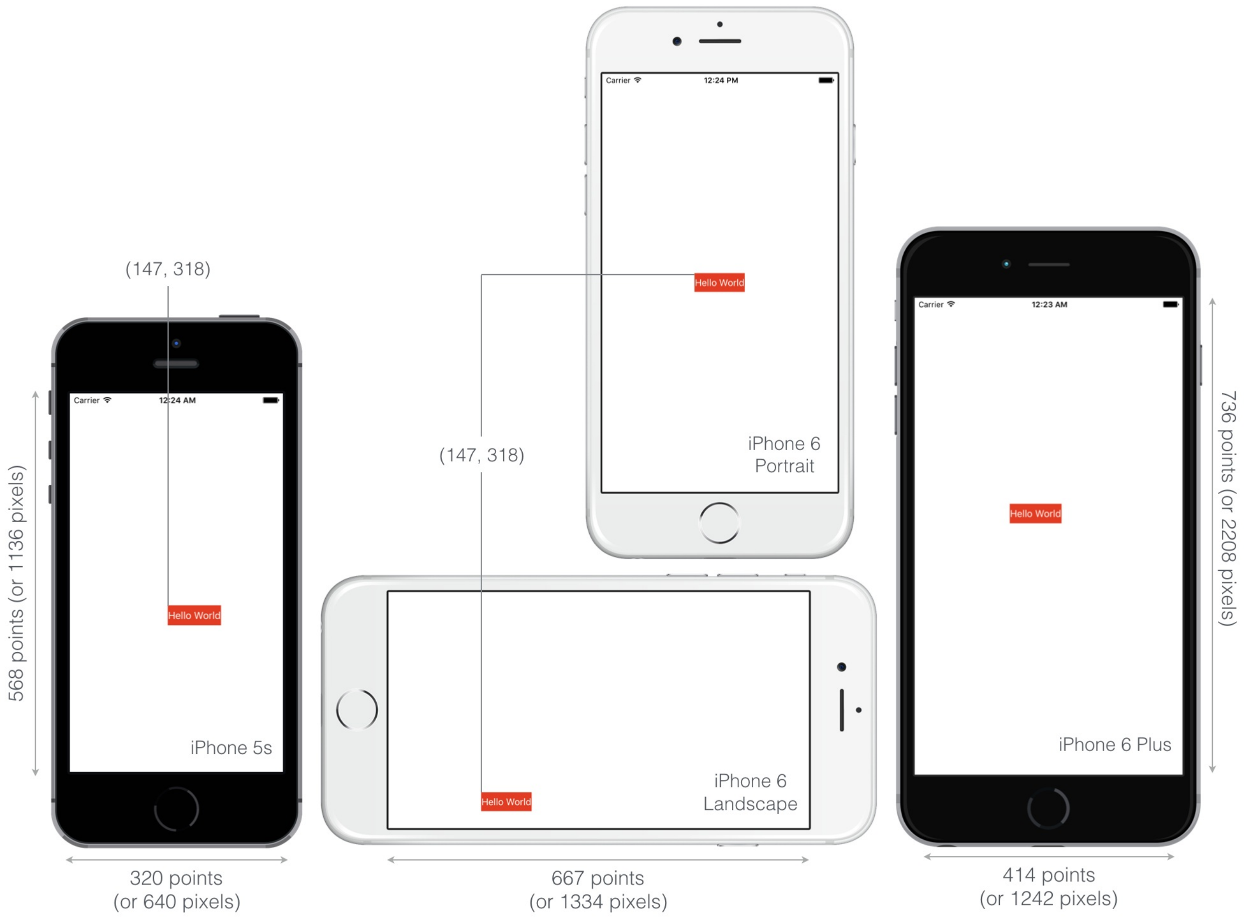
这是为何? 出啥幺蛾子了?

如你所知，iPhone设备有不同的屏幕尺寸:

- 对于iPhone 5/5s, 竖屏模式下水平方向为320 pt(640 px)，竖直方向为667 pt(1334 px)。
- 对于iPhone 6/6s, 水平方向为375 pt(750 px)，竖直方向为568 pt(1136 px)。
- 对于iPhone 6/6s Plus, 水平方向为414 pt(1242 px)，竖直方向为736 pt(2208 px)。
- 对于iPhone 4s, 水平方向为320 pt(640 px)，竖直方向为480 pt(960 px)。

> ## 为何使用点(Point pt)而不是像素(Pixel px)?
>
> 苹果在2007年发布的初代iPhone是3.5英寸的屏幕，分辨率为320×480。横向是320px，纵向是480px。苹果在iPhone 3G与3GS上也使用的这个屏幕分辨率。显然，那时构建的app是一个点对应一个像素。接着苹果就发布了使用retina显示技术的iPhone 4，屏幕分辨率是640×960的两倍，由于使用了retina显示技术，一个点对应着两个像素。
>
> 点系统更加方便开发者的使用。不管屏幕分辨率如何变化，使用数值为基础分辨率的点(iPhone 4/4s是320×480，iPhone 5/5s是320×568)来处理即可，把点和像素的转换工作交给iOS处理。

没有auto layout的话，我们在故事版中放置的按钮就是固定位置的。换句话说，“硬编码”了按钮的起始点。在我们的例子中，把“Hello World”按钮的起始点设为了(147, 318)。因此，不管你使用的是3.5英寸、4英寸，还是5.5英寸的模拟器，iOS都会把按钮画在指定的位置。下图中画出了在不同设备中的按钮，解释了为何“Hello World”按钮仅仅在iPhone 6/6s上是中间位置，其它设备上都偏离了中心，水平朝向上也同样偏离了。



显然，我们想要在所有iPhone设备和朝向上都如我们期望的那样显示，这就是为何要学习使用auto layout，可以解决这种布局的问题。

Auto Layout is All About Constraints

auto layout是一个基于约束(constraint-based)的布局系统，使开发者可以做出自动适配屏幕尺寸与朝向的UI组件，这听起来不错，不过“基于约束的布局”是什么意思？让我来解释一下，想想之前的那个“Hello World”按钮，如果你想把它放在视图中心会怎么描述它的位置？也许你会这么描述：

*不管怎样的屏幕分辨率与朝向，按钮都在水平与竖直方向的中心。

实际上你已经定义了两个约束:

- 水平居中
- 竖直居中

这些约束表示出了界面中按钮布局的规则。

Auto layout的核心就是一些约束，当用文字描述约束时，在auto layout中这些约束会通过数学形式来表示。比如说，当定义一个按钮的位置时，也许想说“按钮的左侧边界应距容器视图的左侧边界30pt。” 变成数学形式就是 button.left = (container.left + 30)。

幸运的是不需要我们去处理这些公式，你只需知道如何使用Interface Builder创建合适的约束即可。

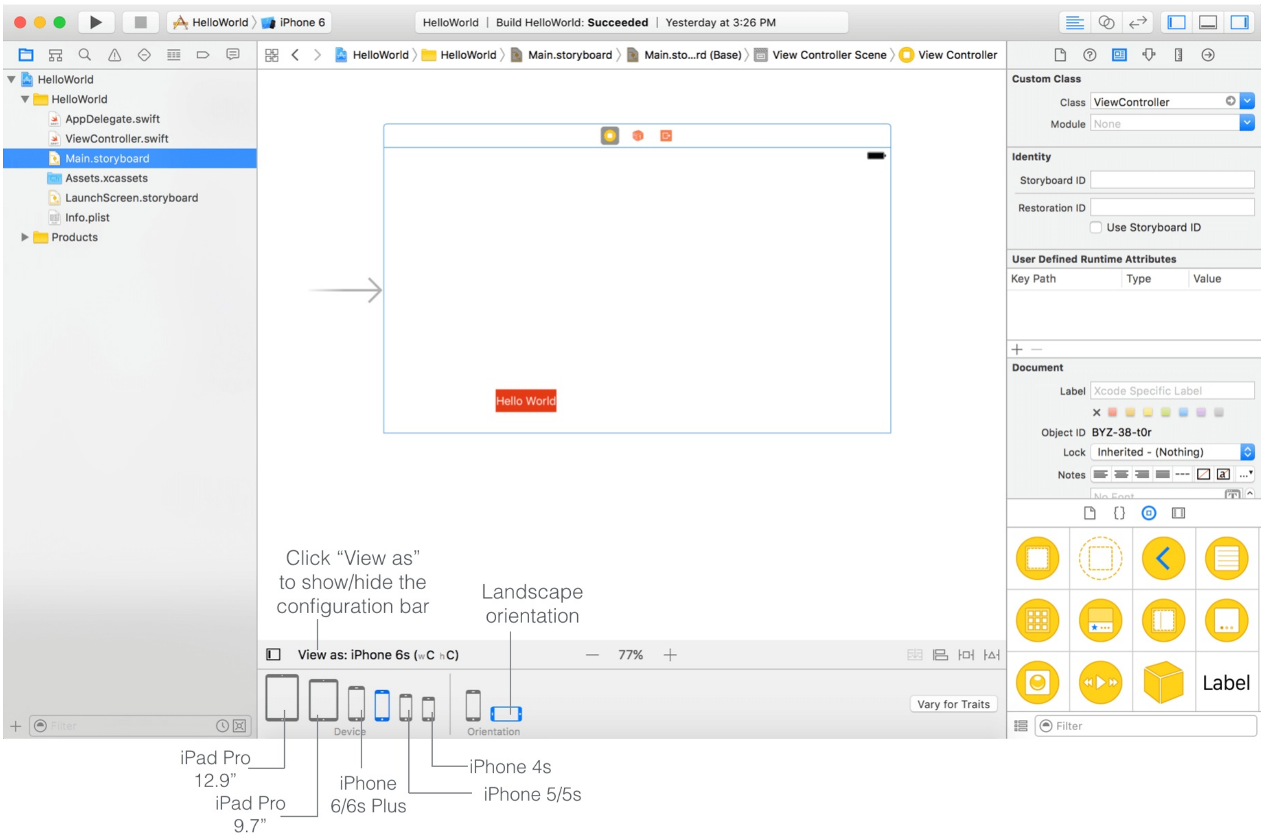
好了，auto layout的基本知识差不多够了，现在来看看在Interface Builder中如何定义处于中心位置“Hello World”按钮的约束。

Interface Builder中的实时预览

首先，打开HelloWorld工程中的Main.storyboard(或者从这儿下载:http://www.appcoda.com/resources/swift3/HelloWorld.zip). 在UI界面上添加布局约束之前，先简要介绍一下Xcode 8中的一个方便的特性。

除了在模拟器中观察app的UI外，Xcode 8还在Interface Builder中提供了一个配置栏，方便开发者进行用户界面的实时预览。

默认情况下Interface Builder被设为预览iPhone 6s的界面，为了观察在其他设备上的效果，点击 `View as: iPhone 6s` 按钮展开配置栏，然后选择你想要测试的iPhone/iPad设备，也可以改变设备的朝向。下图展示了一个Hello World app在iPhone 6s横屏时的实时预览。



配置栏是Xcode 8中的一个很棒的特性，花点时间把玩一下。

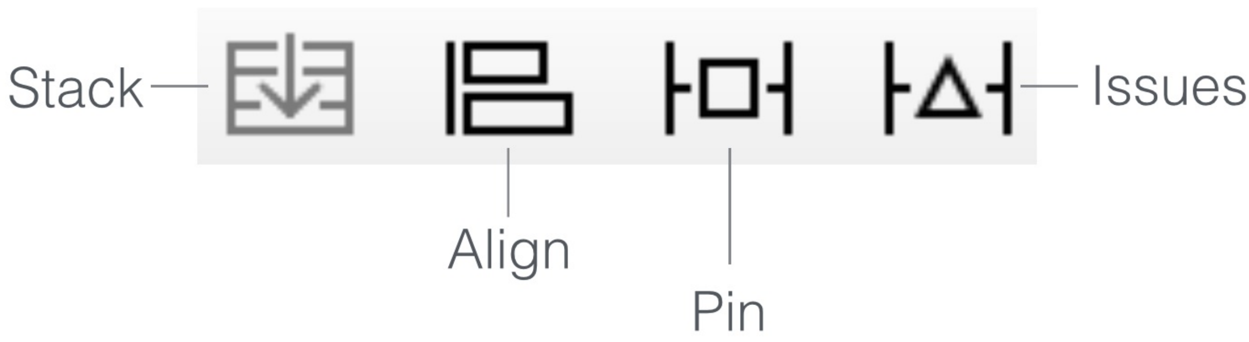
> 注意: 可能会想“(wC hC)”是什么意思，先把它忘掉专注于学习auto layout。若对这个感兴趣可以看下我们的Swift教程。

使用Auto Layout居中按钮

来继续讨论auto layout，Xcode提供了两种定义约束的方法:

1. Auto layout bar
2. Control-drag

在文章中会使用两种方法来进行约束。首先使用auto layout bar，在Interface Builder编辑器的右下角应该会看到四个按钮，这些按钮都属于布局栏，可以使用它们来定义不同类型的布局约束。

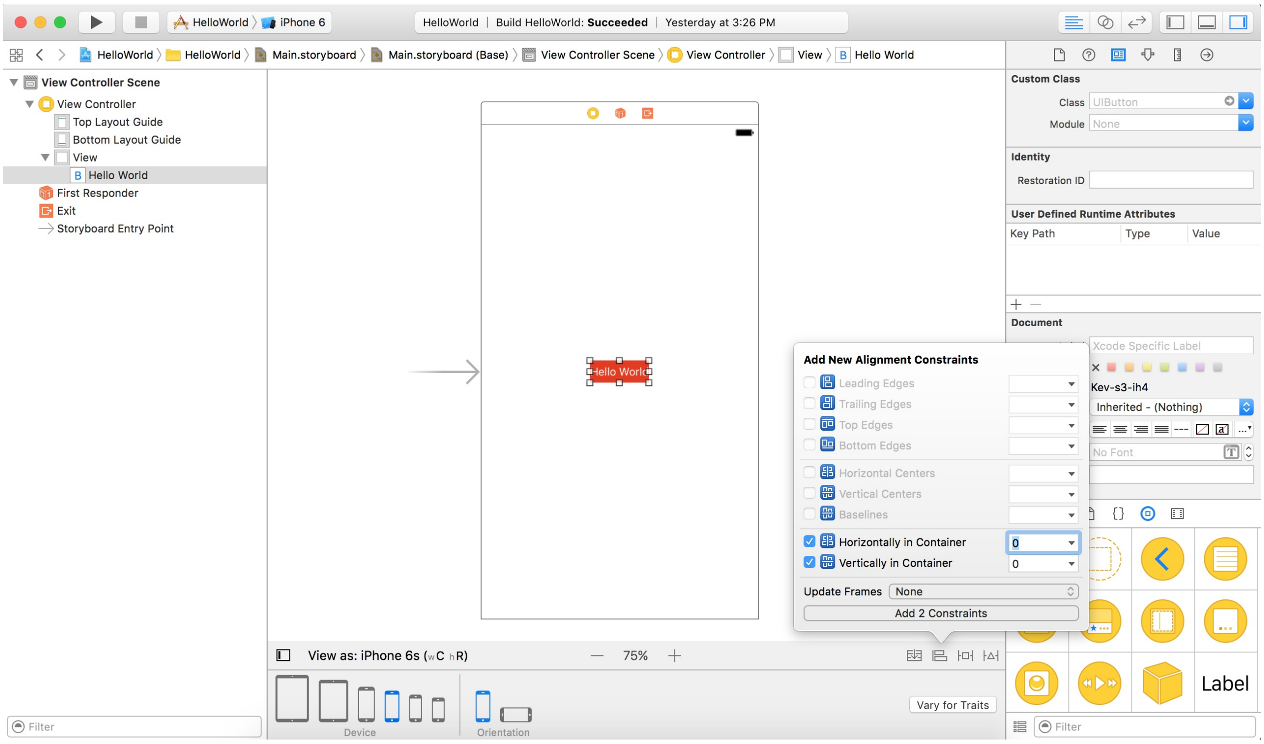


每个按钮都有自己的方法:

- 对齐 – 新建对齐约束，比如对齐两个视图的左边界。
- 固定 – 新建空间约束，比如定义一个UI空间的宽度。
- 问题 – 解决布局问题
- 栈 – 把视图嵌入到栈视图中，栈视图是Xcode 7中出现的新特性，在下一节会讨论这个。

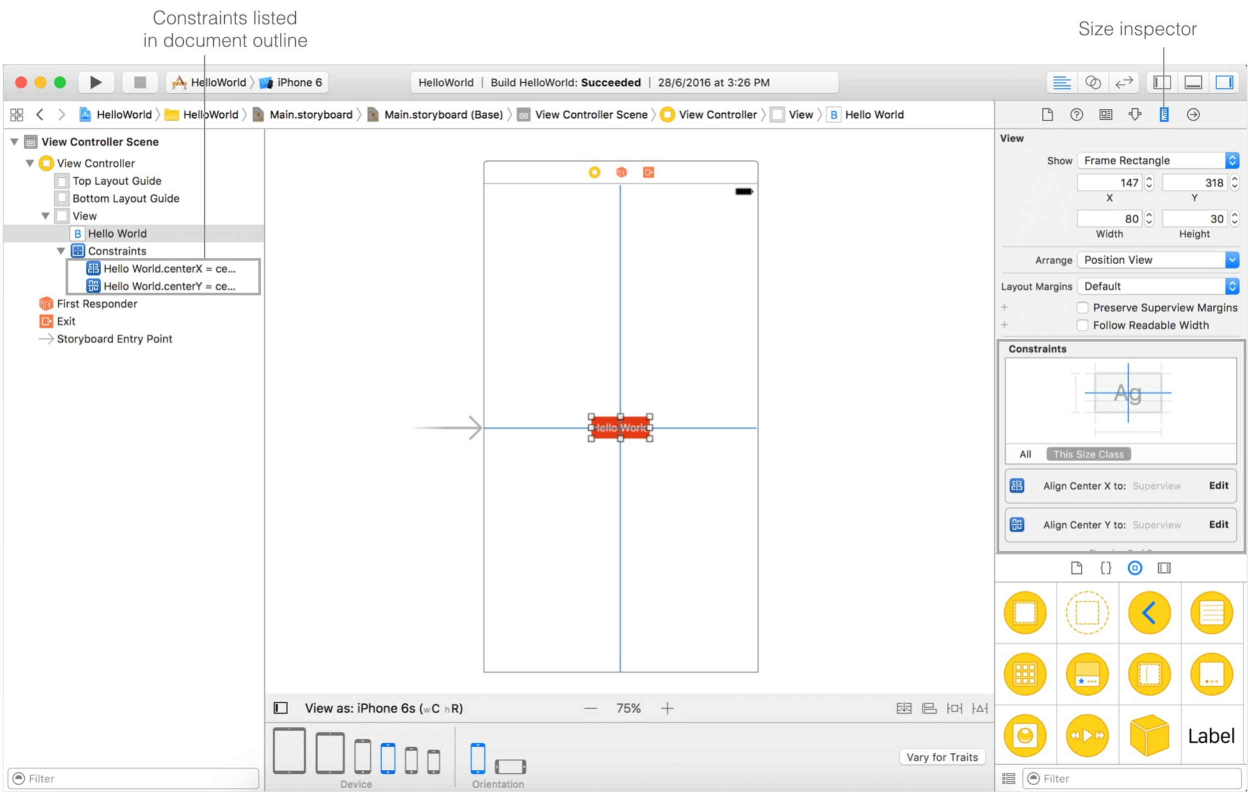
如之前所说的那样，要居中“Hello World”按钮需要定义两个约束：水平居中与竖直居中，这两种约束都是相对于视图的。

需要使用对齐方法来创建约束。首先在Interface Builder选择按钮然后点击布局栏中的对齐图标。在弹出的菜单中，选中“Horizontal in container”和“Vertically in container”选项，然后点击“Add 2 Constraints”选项。



> 提示：可以按下command+0来隐藏项目导航栏，这样可以释放一些屏幕空间，设计起来更方便。

可以看到一些约束线条，展开左侧文档视图的Constraints选项，可以看到刚刚给按钮添加的两个约束，这些约束确保按钮一直保持在视图中心的位置，你也可以在右上角的Size inspector中查看这些约束。



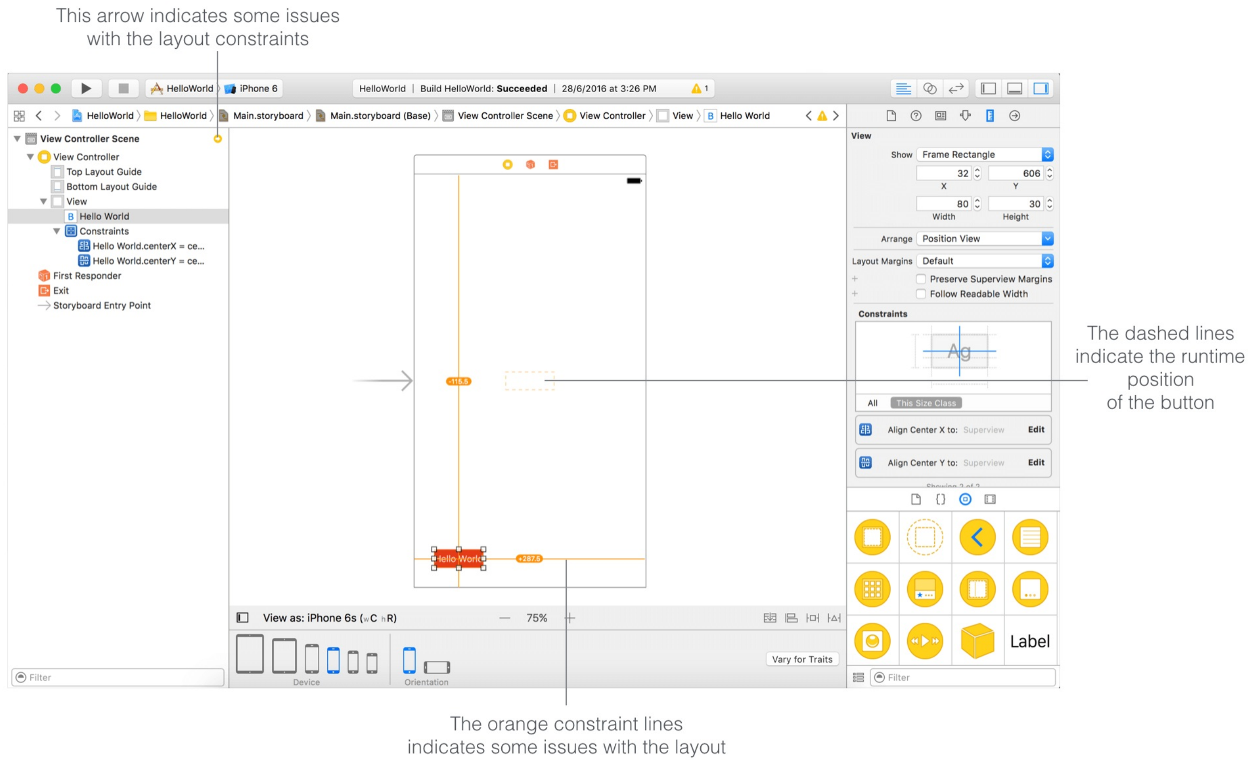
> 提示：蓝色的约束线条表示视图布局的设置是正确无冲突的。

做好测试app的准备了，点击Run使用iPhone 6/6s Plus(或iPhone 4s)模拟器启动app，也可以在底部的配置栏中选择其他设备与设备朝向来检查布局的情况。不论何种屏幕尺寸与朝向，按钮都会完美的出现在屏幕中心。

解决布局约束的问题

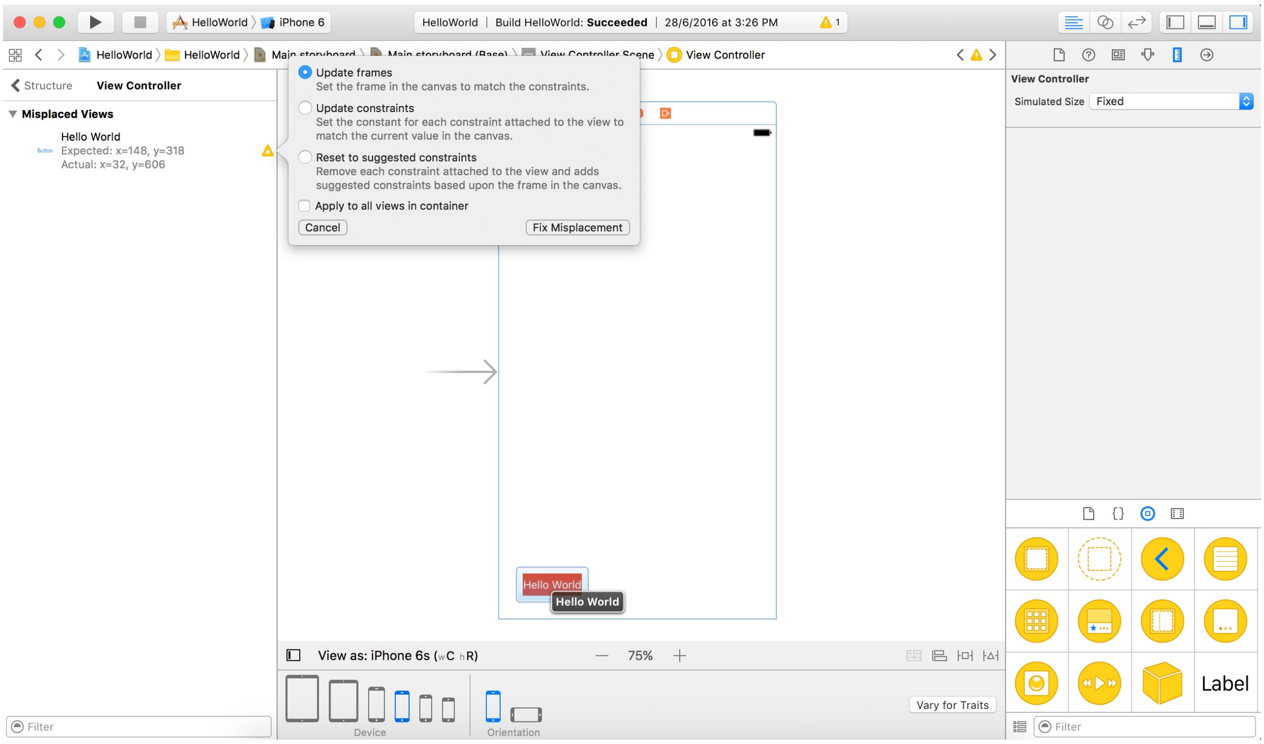
我们刚才设置的布局约束是没问题的，不过总会有出问题的时候，不用担心，Xcode可以检测出这些约束问题。

试着把Hello World按钮拖到屏幕的左下方部分，Xcode会立即检测到布局问题，通过橙色的约束线条指出错位的部件。



当创建的约束不明确或有冲突时会出现自动布局的问题。在这里需要按钮出现在容器的水平与竖直方向的中心，不过按钮现在被放置在了视图的左下角，因此Interface Builder会检测出这个问题并用橙色的线条指出布局问题。

当存在布局问题时，文档大纲视图中会显示一个未闭合的箭头(红/橙)，点击箭头可以看到一个问题列表。Interface Builder会帮我们解决这些问题，点击问题旁的指示图标会出现一些解决方法，选择“Update Frame”并点击“Fix Misplacement”，按钮就被移动到视图中心了。

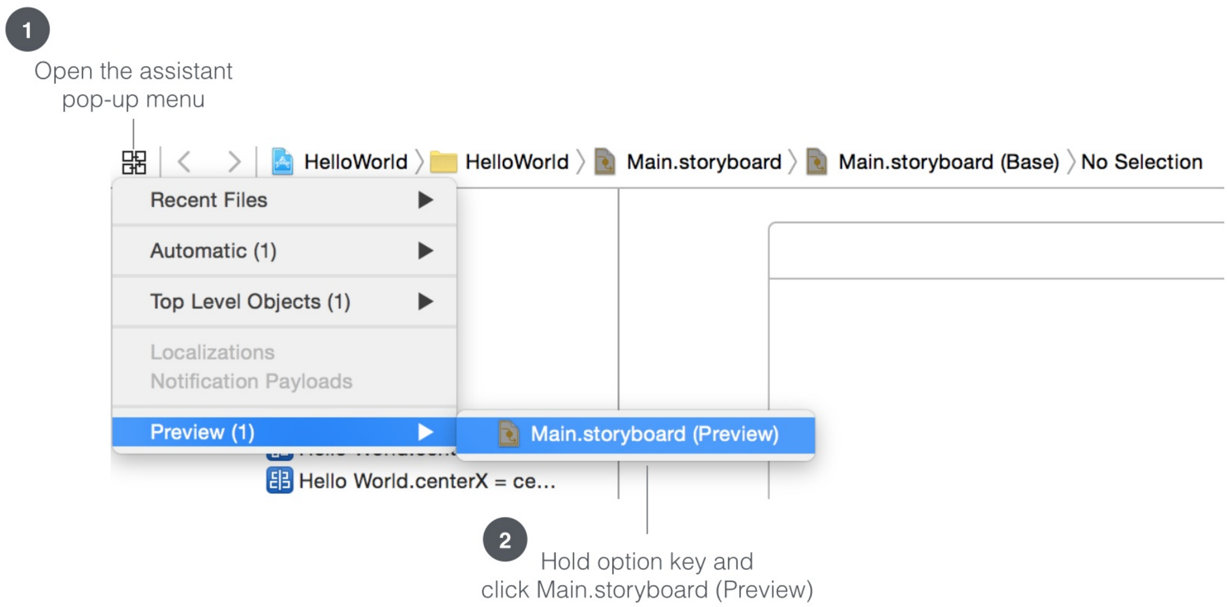


为了示范如何找到并修复问题我是手动触发的这个问题。在之后章节的练习中会遇到相似的布局问题，那时就知道如何快速的解决了。

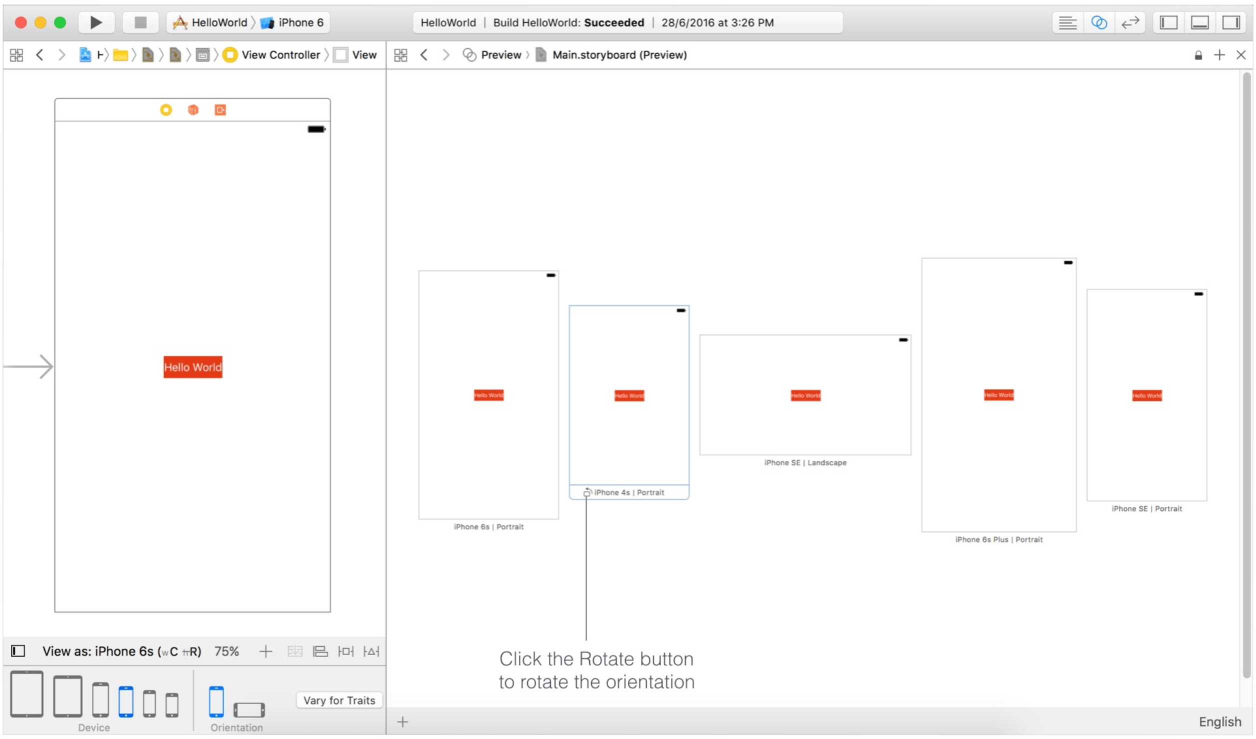
另一种预览故事板的方式

除了配置栏实时预览app的UI界面外，Xcode还为开发者提供了一种在不同设备上预览UI的方法。

在Interface Builder中打开辅助用弹出菜单 > Preview (1)，按住option键的同时点击Main.storyboard (Preview)，如下图：



Xcode会在辅助编辑器中显示app的UI预览界面。默认会使用iPhone 6s来显示。可以点击辅助编辑器左下角的+按钮来添加其它iOS设备(如iPhone 4s/SE)，若想看看屏幕水平朝向时的效果，直接点rotate按钮就行了。预览特性对设计app的UI界面非常有帮助。可以在修改故事板中内容的同时实时观察UI在所选设备中的情况。



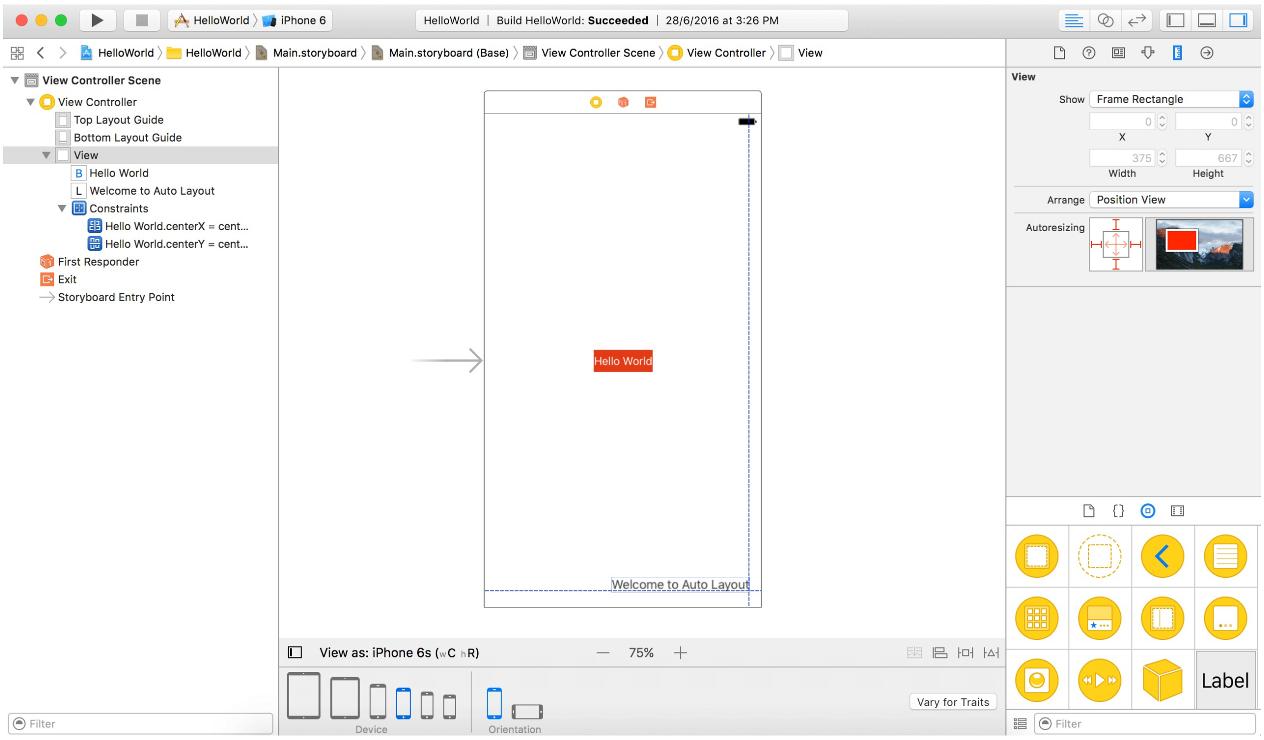
若你想给预览面板腾出更多的屏幕空间，可以在按下command和option的同时按下0来隐藏Utility区域。

> 提示: 如果你在辅助预览中添加了多个设备，那么Xcode也许不能一次性显示所有设备中的预览，有触摸板的话可以使用双指的左划与右划来滚动预览视图，只有鼠标的话可以按下shift的同时使用滚轮进行水平滚动。

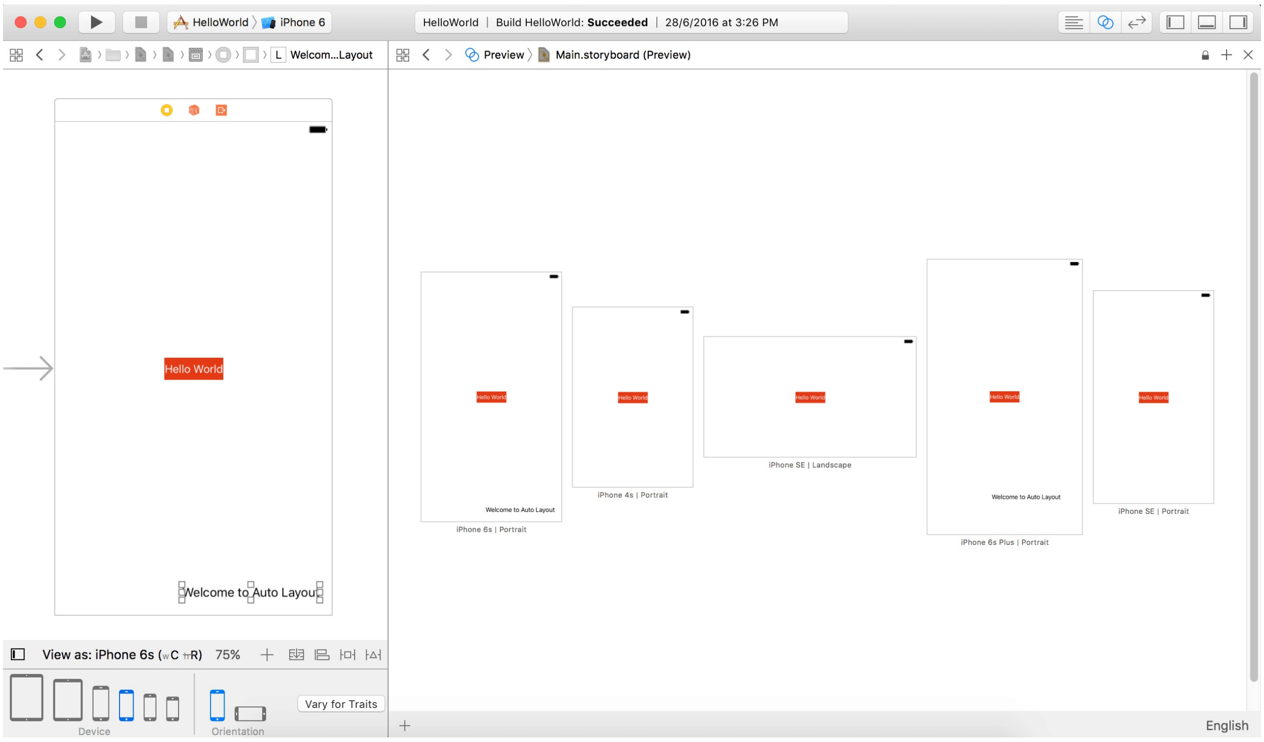
添加标签

现在你已经对auto layout和预览特性有所了解了，来在视图右下角试着添加一个标签吧，看看如何添加标签的布局约束。iOS中的标签常被用来显示简单的文本与信息。

从Interface Builder编辑器的对象库中拖一个标签到视图的右下角。双击标签把文本改成“Welcome to Auto Layout”或你想要的内容。



若再次打开预览辅助，会看到UI界面立马发生了变化。在没有定义标签的布局约束的情况下，在除了iPhone 6s/6s外设备上没有显示出标签。



如何解决这个问题？显然，需要设置一组合适的约束，问题是：应该添加怎样的约束？

来尝试用文字描述一下标签的位置要求，也许会这么表达：

*标签应被放置在视图的右下角。

这么描述可以不过不够具体，下面这样描述标签的位置更加具体：

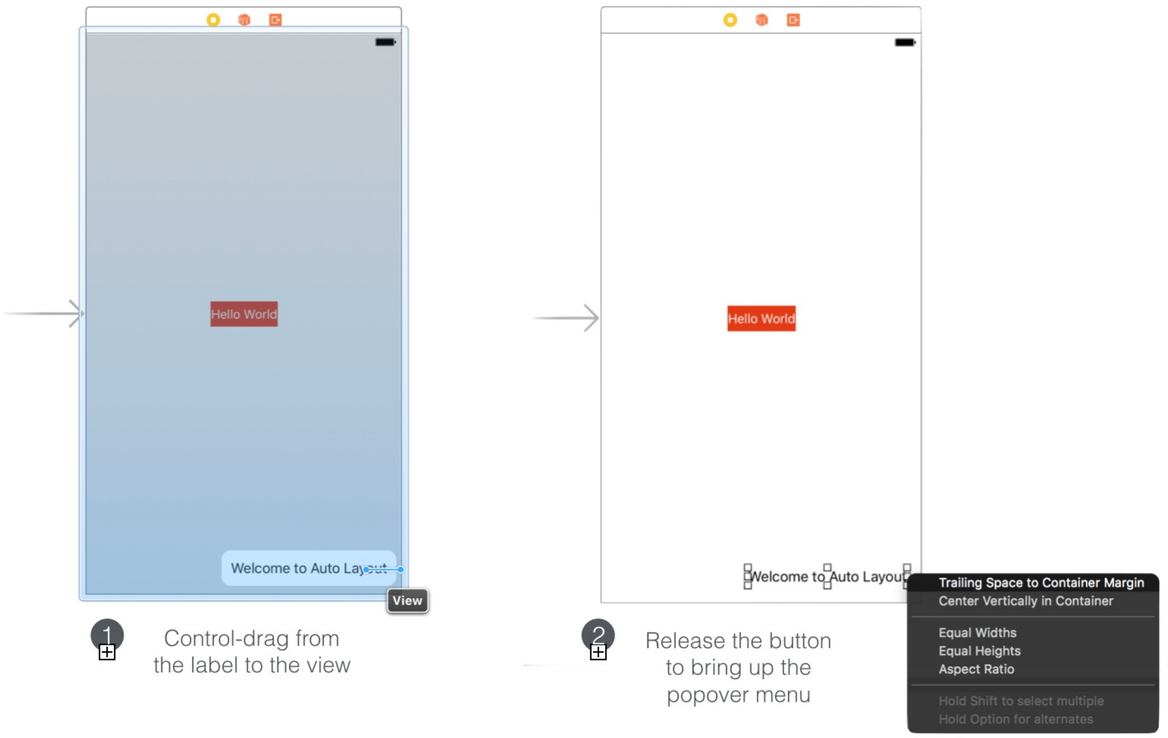
*标签距离视图右边界0 points，距离下边界20 points。

这就好多了，能具体地描述一个组件位置的话，就可以轻松添加布局约束了。那么标签的约束如下:

1. 标签离视图右边界的内距为0 points。
2. 标签离视图下边界的内距为20points。

我们将这些约束转化为auto layout中的空间约束，可以使用布局按钮中的Pin来创建空间约束，不过这次我们使用Control-drag的方式来进行auto layout。在Interface Builder中，可以在使用control键的同时将一个部件拖拽到你想要添加约束的另一个部件的坐标轴上。

来添加第一个空间约束，按住control键把标签拖拽到右侧直到视图变成蓝色高亮的状态，松开按钮后会出现一个包含约束列表的菜单。选择“Trailing space to container margin”添加一个按钮与视图右边界的空间约束。

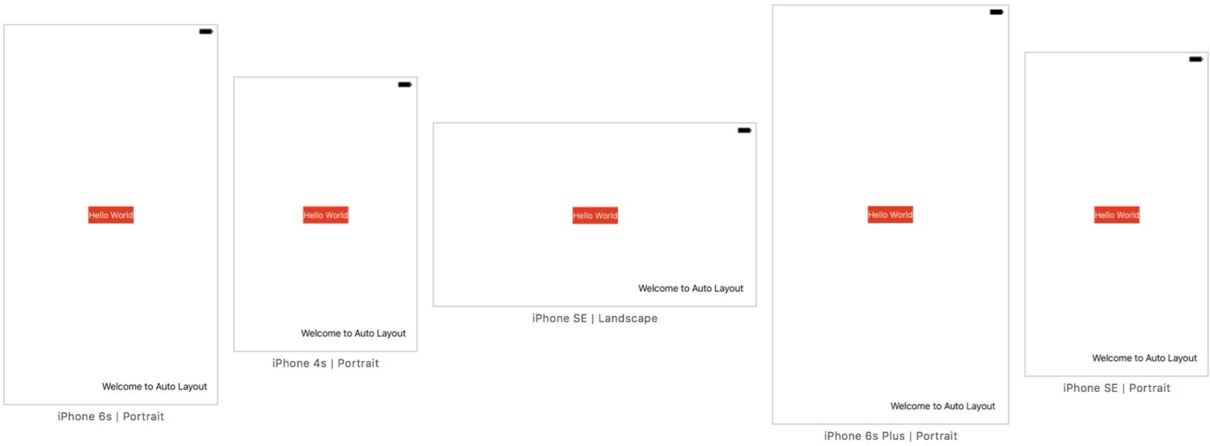


在文档大纲视图中可以看到新的约束。Interface Builder用红色的线条指示出缺少了一些约束，这是正常的，因为还没有定义第二个约束。

现在按住control把标签拖拽到视图底部，松开按钮选择“Vertical Spacing to Bottom Layout Guide”，创建一个按钮与视图下边界的空间约束。



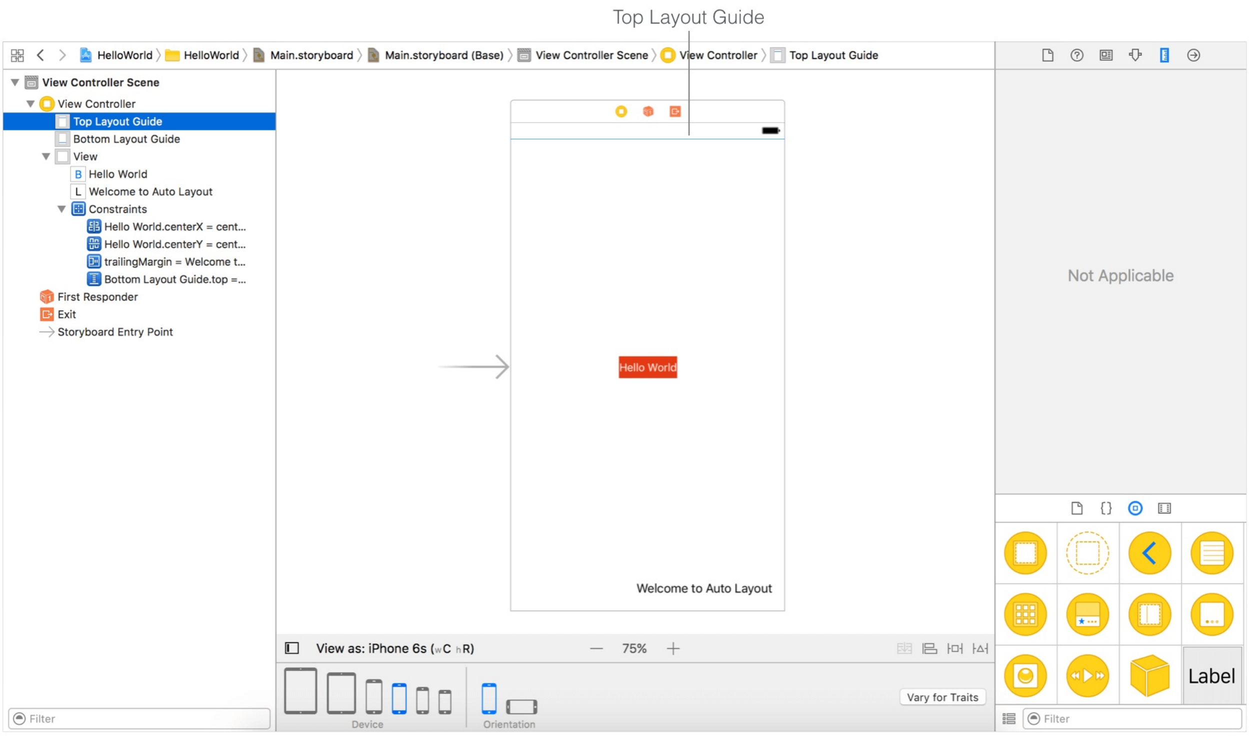
添加完这两个约束后，所有的约束线条都应该是蓝色的实线。在预览UI或在模拟器中运行app时，会看到不管是何种尺寸、何种朝向的屏幕，标签都出现了正确的地方。



Top 和 Bottom Layout Guide

Bottom Layout Guide(底部布局辅助)是啥东东呢？通常来说，Bottom Layout Guide是与底部视图关联的一个约束边界，有时会发生变化，比如存在一个工具栏的话，Bottom Layout Guide就会与工具栏的顶部关联。

Top Layout Guide的话是在距视图顶部20points(状态栏的高度)的位置。



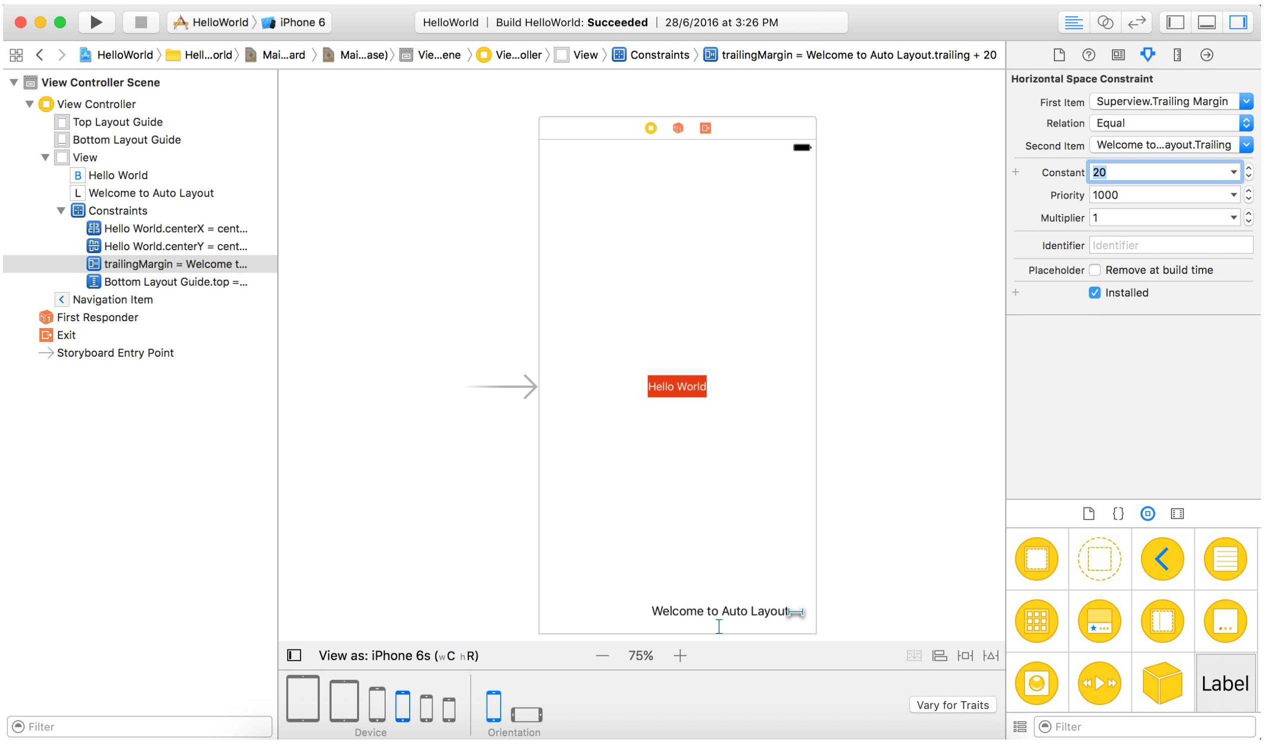
这些布局辅助约束对于定义布局约束来说是非常有用的，因为他们会根据视图的布局变化自动调整。比如说，如果视图中有一个导航栏，那么top layout guide就会移动到导航栏的下方(如下图)。因此，只要UI组件的约束是与top/bottom layout guide相关联的，那么在添加一个导航栏或工具栏时界面就会自动调整成正确的布局。T



编辑约束

“Welcome to Auto Layout”标签现在被放在了距离视图右边界0 points的地方，若要在标签和视图右边界中添加一些空间怎么办呢？Interface Builder提供了一种很方便的修改约束的方法。

首先选择文档大纲视图中的约束，这里你可以选择“trailingMargin”约束。在属性查看器中找到这个约束的属性，比如relation, constant和priority。constant现在是0，可以改为20添加一些额外的空间。

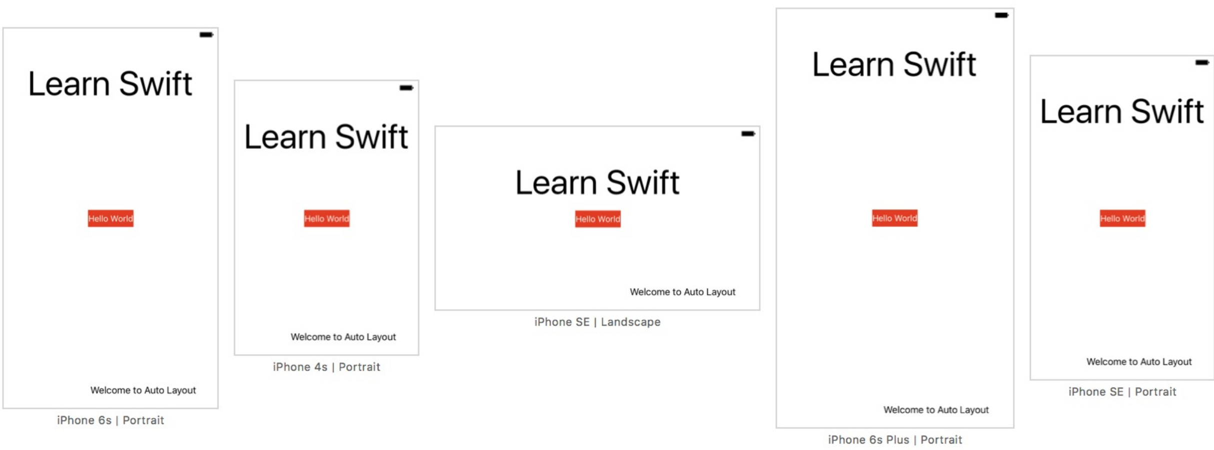


练习

我觉得你现在对适配各种屏幕尺寸的方法应该有一定的了解了，来进行一个简单的练习吧。需要你做的是再添加一个名为Learn Swift的标签到视图中，并满足如下约束：

- 新标签距离top layout guide 40 points。
- 新标签在水平中心位置。

可以把标签的字体大小调整为30points，打开属性观察器编辑Font选项即可，结果应如下所示：



总结

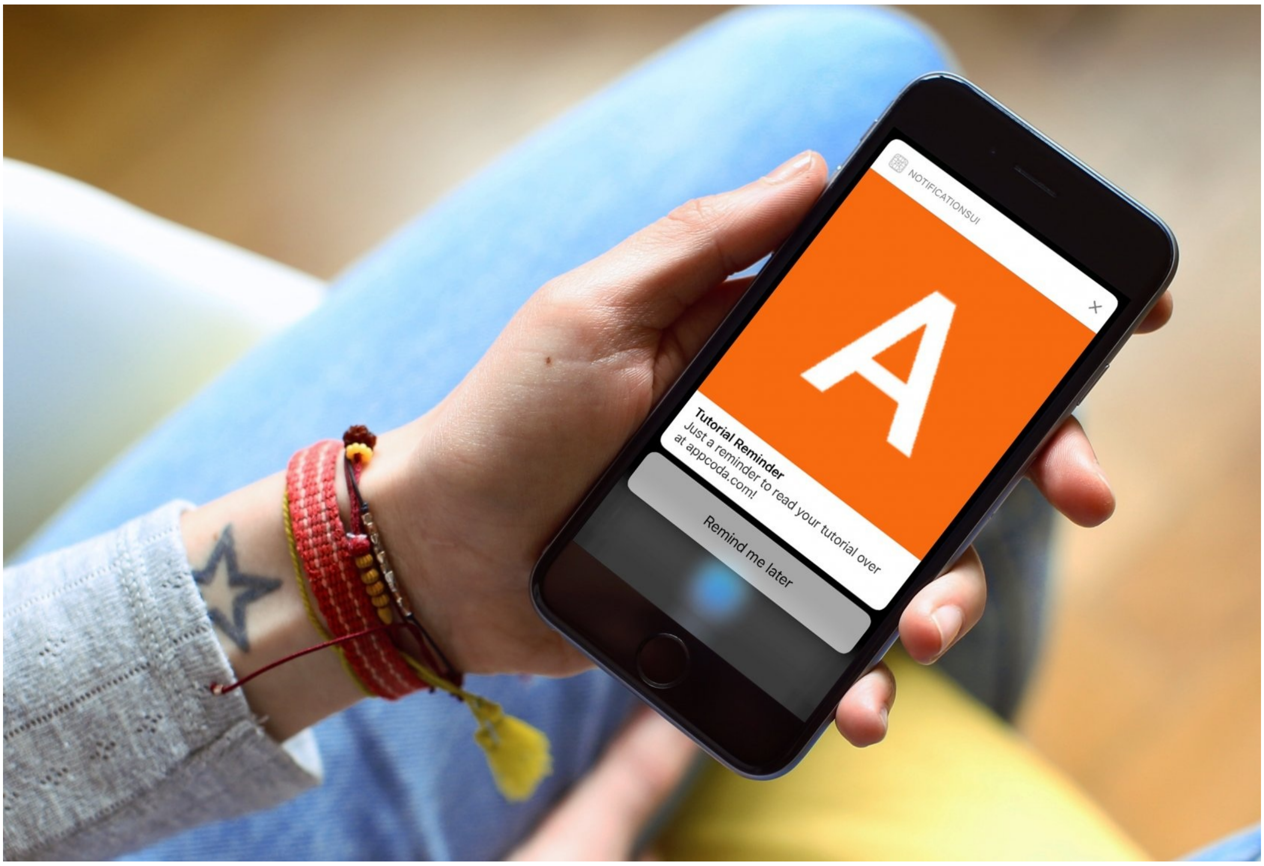
在这里只是介绍了一下auto layout的基本内容，在开发app的过程中再更深的去探索一下它的奥秘吧！

可以在这里下载完整项目

Introduction to User Notifications Framework in iOS 10

iOS10中的User Notifications框架介绍

> - 原文链接：Introduction to User Notifications Framework in iOS 10
> - 原文作者：PRANJAL SATIJA
> - 译者：yrq110

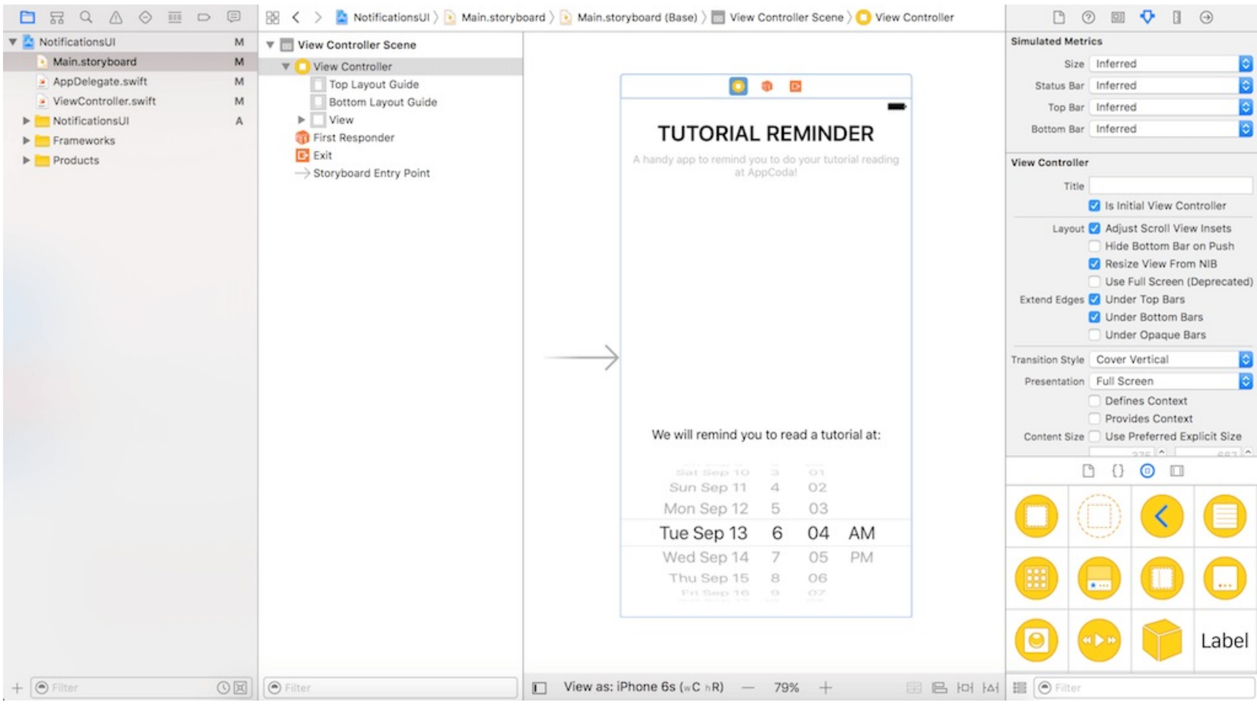


欢迎阅读iOS 10中的推送教程，今天我们要讲的是如何在iOS10中实现推送，iOS10在推送相关的API上做了大量的改变，包括一个全新、强力的给予触发器的推送请求系统，会简要介绍一下这个系统，此外改变最大的部分就是一个叫做UserNotificationsUI的框架，它允许开发者将自定义的视图控制器嵌入到推送中！这就使我们可以创建出以前无法实现的强有力的推送。

这篇教程的内容中涵盖iOS中不同类型的本地推送，会从一个基本的文本推送开始，之后逐渐在推送中加入图像与用户操作。

简单用户推送

并不是所有我们想展现的推送都是漂亮与可交互的，因此从一个基本的文本推送开始，下载开始项目，打开NotificationsUI.xcodeproj。浏览一下项目内容：设计用户界面的Main.storyboard，ViewController.swift，还有AppDelegate.swift。



打开Main.storyboard并浏览下ViewController中的内容，今天所要做的app具有提醒用户阅读AppCoda文章的功能，会让用户选择一个推送的日期与时间。为了达到目的，第一个任务显而易见：启用推送。现在选择日期与时间没有任何效果，需要修改app来设置一个日期时间使其在设定的时间进行推送。

打开ViewController.swift，开始吧！

ViewController中没什么代码，只有一个名为datePickerDidSelectNewDate的方法，当在日期选择器中选择日期后会调用这个方法。如前面所说的那样，现在app只需要创建一个推送即可，来一起实现这个功能。

iOS 10中推送的创建方法稍有改变，开发者必须先创建请求，请求iOS显示一个推送。这些请求由两部分组成：触发器和内容。为了在iOS 10中显示推送，需要从创建一个触发器开始，要使用一个基于日期的触发器——UNCalendarNotificationTrigger类。苹果同时也提供了其它一些触发器，比如UNLocationNotificationTrigger(当用户到达一个特定的地点后触发推送)。

为了在iOS中显示推送，第一步需要创建触发器，Let's go。

打开AppDelegate.swift文件，确保引入了UserNotification框架，所有与推送有关的API都在这个框架中:

```
import UserNotifications
```

接着添加scheduleNotification(at date: Date)方法，顾名思义，这个方法根据提供的日期来安排推送的时间。

```
func scheduleNotification(at date: Date) {
    let calendar = Calendar(identifier: .gregorian)
    let components = calendar.dateComponents(in: .current, from: date)
    let newComponents = DateComponents(calendar: calendar, timeZone: .current, month:
components.month, day: components.day, hour: components.hour, minute: components.minut
e)
}
```

在上述代码中使用公历日历将提供的日期分割为不同要素(年、月、日、小时、分钟)。这部分内容超出了教程的范围，不过你需要知道一个日期是由要素组成的。一个独立的要素表示日期中的一部分，比如小时、分钟、秒等。代码将日期分割为不同要素并保存在常量中，这么做是因为UNCalendarNotificationTrigger需要日期要素，而不是日期，有一点奇怪的是，UNCalendarNotificationTrigger不接受一个由日期中所有要素组成的输入，因此需要自己通过已存在的对象建立DateComponents对象。方法中的第三行代码就是那么做的：使用日期中的相关信息来创建DateComponents对象。

现在获得了日期中的要素，可以来制作推送触发器了，这一步很简单，如下添加代码即可:

```
let trigger = UNCalendarNotificationTrigger(dateMatching: newComponents, repeats: false
)
```

如你所想，这行代码使用刚刚提取的日期要素新建一个UNCalendarNotificationTrigger对象，将repeats属性设为false使其只出现一次。

创建好触发器了，接下来创建想在推送中显示的内容，使用UNMutableNotificationContent类来做这件事。添加如下代码来创建推送的内容，把代码放在触发器变量的后面即可:

```
let content = UNMutableNotificationContent()
content.title = "Tutorial Reminder"
content.body = "Just a reminder to read your tutorial over at appcoda.com!"
content.sound = UNNotificationSound.default()
```

上面的代码新建了一个UNMutableNotificationContent对象，然后设置它的title，body和sound属性。

看起来已经做好显示第一个推送的准备了！还有些事儿没做：1.创建一个推送请求。2.提交给系统，告诉iOS显示我们的推送。

需要使用触发器与内容来初始化一个UNNotificationRequest对象，还需要给予请求一个唯一标识符。在scheduleNotification函数中插入如下代码:

```
let request = UNNotificationRequest(identifier: "textNotification", content: content,
trigger: trigger)
```

小case，不是吗? 使用了三个参数来创建请求:

- 标识符: 请求的唯一标识符，这个标识符可以用来取消推送请求。
- 内容: 之前创建的推送内容。
- 触发器: 用来触发推送的触发器，当满足条件时会iOS会显示推送。

好了，最后一件事就是将请求添加至推送中心，推送中心管理着app中的所有推送，推送中心会监听相应的事件并在合适的时机触发推送。

在将请求添加至推送中心前，最好先移除已有的推送请求。这一步可以避免重复的推送请求，在函数中插入如下代码段:

```
UNUserNotificationCenter.current().removeAllPendingNotificationRequests()
UNUserNotificationCenter.current().add(request) {(error) in
    if let error = error {
        print("Uh oh! We had an error: \(error)")
    }
}
```

第一行代码取得推送中心中的所有对象后，调用removeAllPendingNotificationRequests方法移除所有待定的推送请求，接着将请求添加到推送中心。这个方法会接收一个完成后的handler，使用handler输出错误信息。在开始前确保你的scheduleNotification函数如下所示:

```
func scheduleNotification(at date: Date) {
    let calendar = Calendar(identifier: .gregorian)
    let components = calendar.dateComponents(in: .current, from: date)
    let newComponents = DateComponents(calendar: calendar, timeZone: .current, month:
components.month, day: components.day, hour: components.hour, minute: components.minut
e)

    let trigger = UNCalendarNotificationTrigger(dateMatching: newComponents, repeats:
false)

    let content = UNMutableNotificationContent()
    content.title = "Tutorial Reminder"
    content.body = "Just a reminder to read your tutorial over at appcoda.com!"
    content.sound = UNNotificationSound.default()

    let request = UNNotificationRequest(identifier: "textNotification", content: conte
nt, trigger: trigger)

    UNUserNotificationCenter.current().removeAllPendingNotificationRequests()
    UNUserNotificationCenter.current().add(request) {(error) in
        if let error = error {
            print("Uh oh! We had an error: \(error)")
        }
    }
}
```

现在需要配置ViewController.swift，当用户选择日期时调用cheduleNotification(date:)函数。在ViewController.swift的datePickerDidSelectNewDate方法中插入如下代码:

```
let selectedDate = sender.date
let delegate = UIApplication.shared.delegate as? AppDelegate
delegate?.scheduleNotification(at: selectedDate)
```

上述代码取得当前应用的AppDelegate对象，使用之前缩写的函数设置一个推送。来试一试，运行app并选择一个时间，等待时钟到达那个时刻看看会发生什么!

哇哦! 推送并没有出现，为撒子?

它没有正常工作的原因有如下几点:

首先，苹果力求在iOS中保持平滑的用户体验，给予用户对来自app推送的最终控制权就是用户体验的一部分。用户还未授权我们显示推送，这就是为何没有将信息传递过去。

来修复这个问题，在AppDelegate.swift中添加如下方法:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOpt
ions: [UIApplicationLaunchOptionsKey : Any]? = nil) -> Bool {

    UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .sound])
 {(accepted, error) in
        if !accepted {
            print("Notification access denied.")
        }
    }

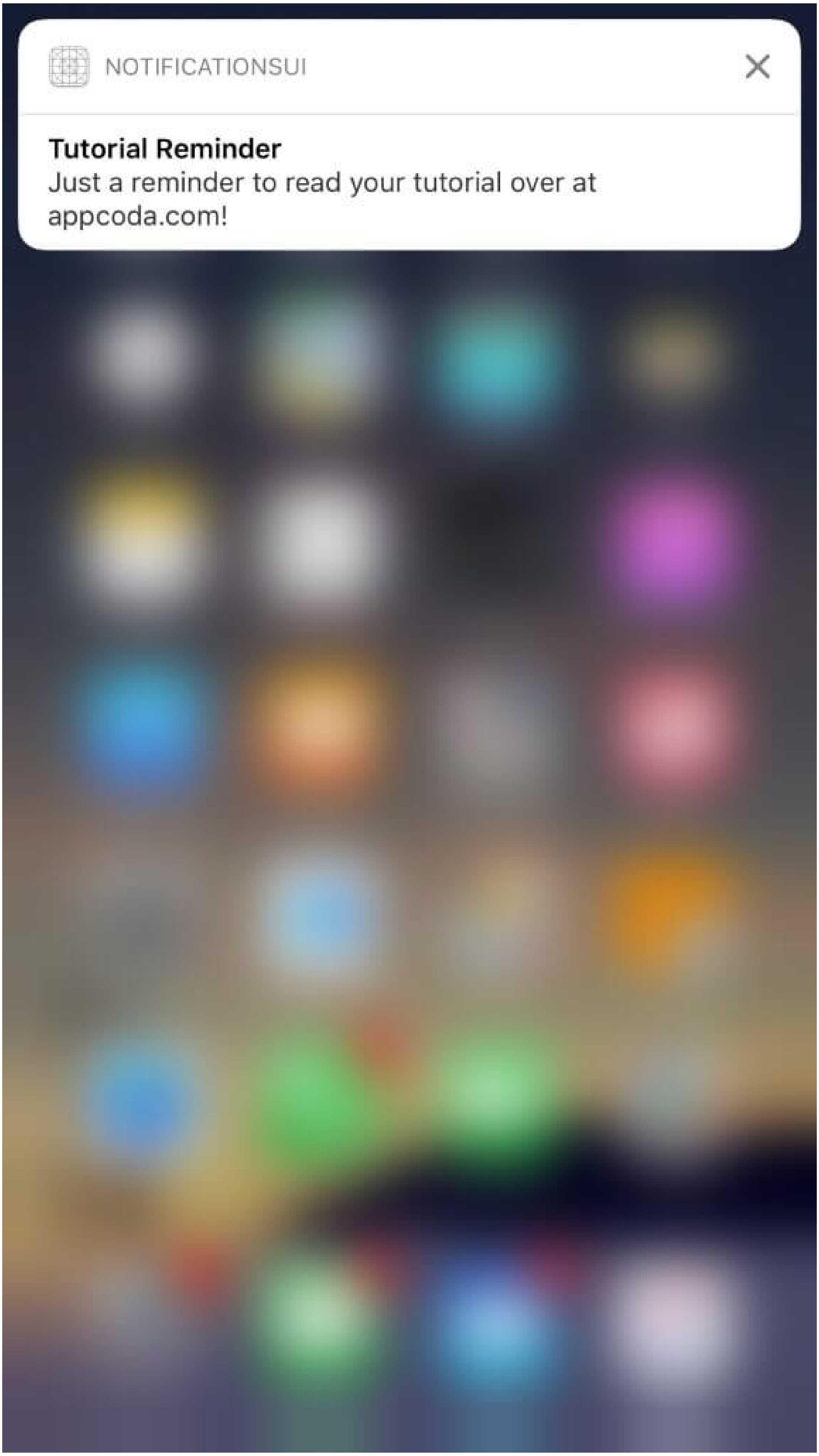
    return true
}
```

这里调用了推送中心的requestAuthorization方法来从用户那里获取关于使用推送的许可，还需要请求显示警告视图和播放音频的权限。

再次运行app。当app启动时，会看到授权请求。点击接受，这样app就被允许给设备发送推送。

需要注意一点的是推送不会在app中显示。，此在设置完日期后确保回到home界面或锁屏界面中。

如果都按部就班了，应该会收到推送！


NOTIFICATIONSUI
Tutorial Reminder
Just a reminder to read your tutorial over at appcoda.com!

看到没？在iOS中给用户推送并不难，只需要做一些前置工作，比如请求推送许可。现在已经可以发送简单文本了，下一步来附上一个图片吧！

在推送中附上一张图片

现在的推送是纯文本的，并没有什么新意。来尝试下其他特性，比如在推送中显示一张图片。

在开始项目中有一张图片: logo.png，没找到请看看NotificationsUI文件夹。来看看如何将它显示在推送中。

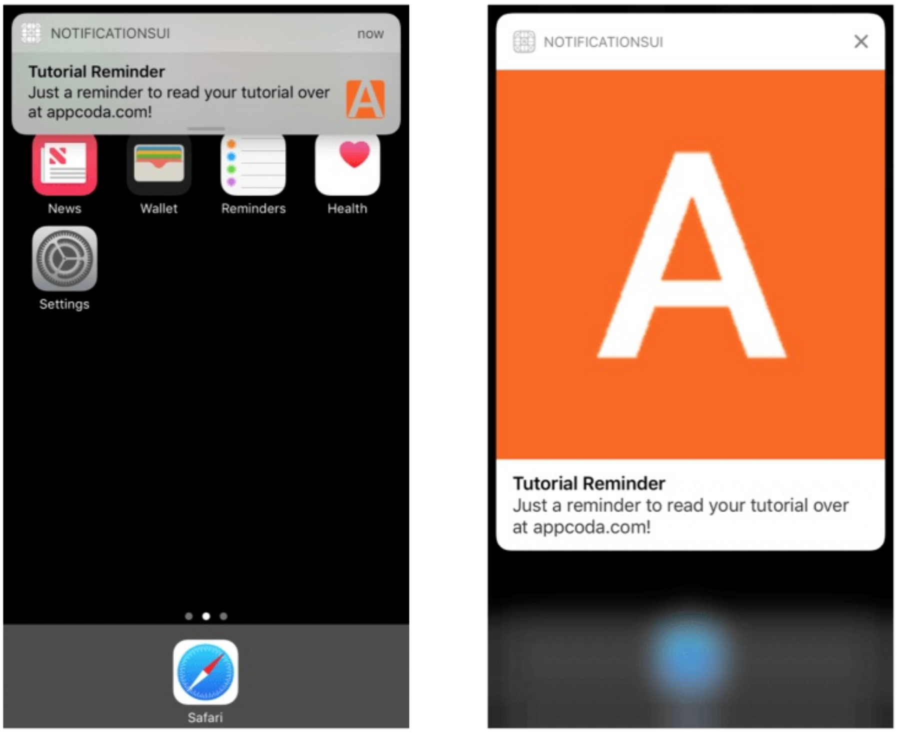
UserNotification框架提供了UNNotificationAttachment类，使用它可以将附件添加到推送中，附件可以时音频、图片和视频，支持多种文件格式。可以在苹果开发者网站上查看详情。

跟UserNotifications框架中的其他任务一样，创建附件也是很简单的。先初始化一个UNNotificationAttachment对象，将其添加到推送内容中即可。修改一下scheduleNotification方法并把如下代码插入到请求变量之前:

```
if let path = Bundle.main.path(forResource: "logo", ofType: "png") {
    let url = URL(fileURLWithPath: path)

    do {
        let attachment = try UNNotificationAttachment(identifier: "logo", url: url, op
tions: nil)
        content.attachments = [attachment]
    } catch {
        print("The attachment was not loaded.")
    }
}
```

上述代码通过文件路径加载logo图片，使用图片初始化一个附件。创建成功后添加到内容中。再次测试app，加载后选择一个日期等待推送出现。



哇哦! 看到了吧，发送了一个带有图片的推送。这是iOS 10中首次引入的新特性，很酷吧，不过我觉得可以做得更好，添加一个"稍后提醒我"的按钮，允许用户暂时忽视提醒。

来一起实现吧。

在推送中添加交互行为

在推送中添加交互行为有一些复杂，需要设置一些API。不过没有我们搞不定的，在推送中添加交互行为需要用到UNNotificationAction和UNNotificationCategory类。

首先要定义action(动作)的category(目录)，简单起见只定义一个category和action。将如下代码添加到application(_:didFinishLaunchingWithOptions:)方法中:

```
let action = UNNotificationAction(identifier: "remindLater", title: "Remind me later",
 options: [])
let category = UNNotificationCategory(identifier: "myCategory", actions: [action], int
entIdentifiers: [], options: [])
UNUserNotificationCenter.current().setNotificationCategories([category])
```

上述代码创建了一个action和category，并在UNUserNotificationCenter中注册了category。

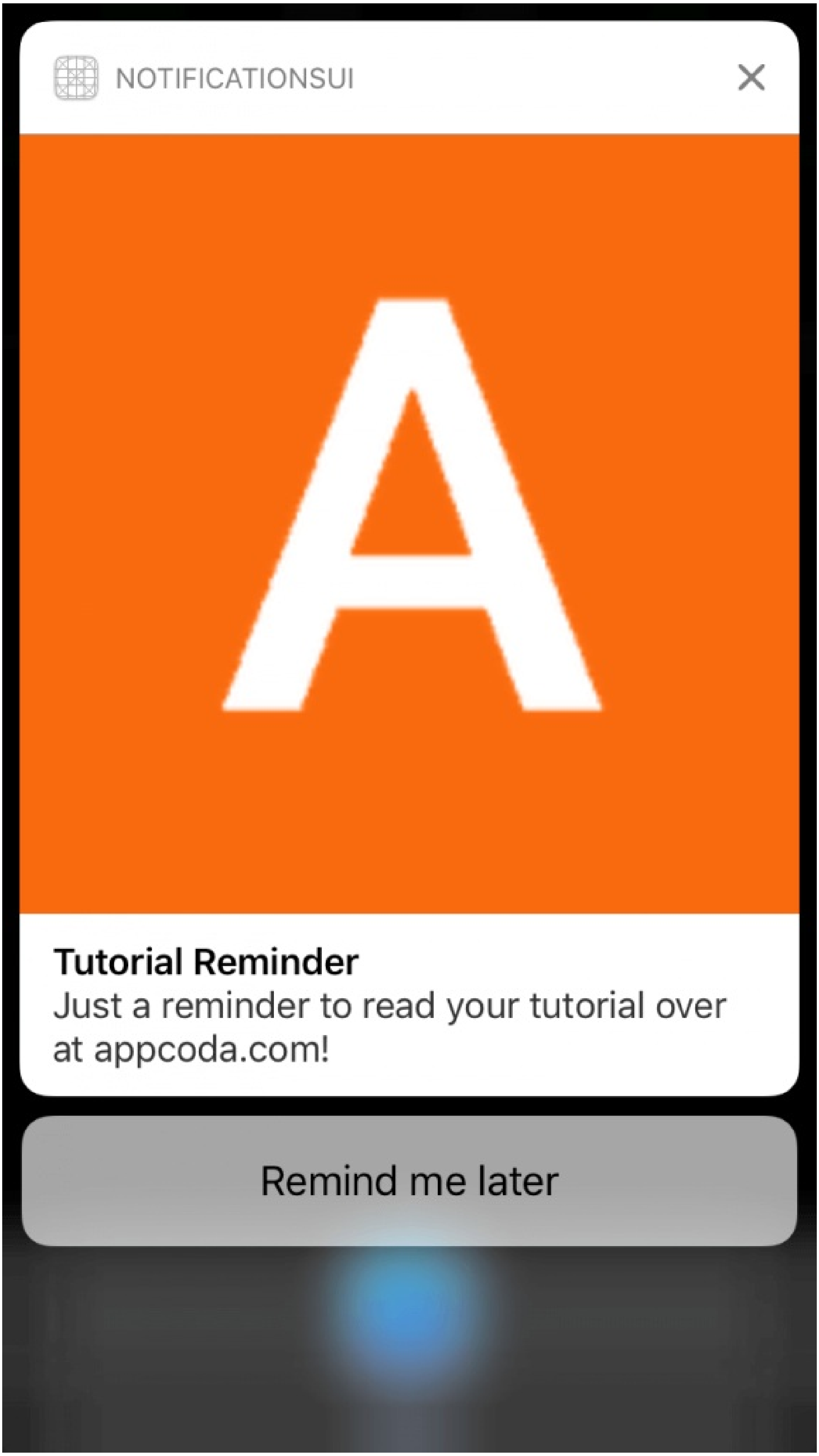
也许你会想为何会有category，它们与action有不同的作用，将category想象成包含很多action的集合。所有在category中包含的action都会体现在推送内容中。

在这个demo中，catrgory中只有一个action。若category中包含多个action的话，将其分配给推送，则推送会使用category中的所有action，这就是为何苹果的工程师会提供UNNotificationCategory类来管理推送中的action。

现在已经创建了action与category并注册了category，接下来在内容中使用它。修改创建内容的代码，添加如下行:

```
content.categoryIdentifier = "myCategory"
```

它会告诉系统将在新的推送中使用“myCategory”目录，再次构建并运行app看看会发生什么:


NOTIFICATIONSUI
Tutorial Reminder
Just a reminder to read your tutorial over at appcoda.com!
Remind me later

棒极了!推送中出现了一个可爱的按钮。不过点击按钮的话只是返回了推送，剩下的任务就是要实现响应动作的代码。

UNUserNotificationCenterDelegate协议定义了推送中行为的响应方法。因此为了响应操作，需要扩展AppDelegate实现UNUserNotificationCenterDelegate协议:

```
extension AppDelegate: UNUserNotificationCenterDelegate {
    func userNotificationCenter(_ center: UNUserNotificationCenter, didReceive respons
e: UNNotificationResponse, withCompletionHandler completionHandler: @escaping () -> Vo
id) {

    }
}
```

当用户点击按钮时会调用userNotificationCenter(_:didReceive:withCompletionHandler:)方法，可以根据输入的UNNotificationResponse对象的actionIdentifier属性判断用户进行的是哪项操作，像下面这样实现这个方法:

```
func userNotificationCenter(_ center: UNUserNotificationCenter, didReceive response: U
NNotificationResponse, withCompletionHandler completionHandler: @escaping () -> Void)
{

    if response.actionIdentifier == "remindLater" {
        let newDate = Date(timeInterval: 900, since: Date())
        scheduleNotification(at: newDate)
    }
}
```

上面的代码中安排了一个距当前时刻900秒(15分钟)后的推送。最后，在scheduleNotification(at:)方法中将推送中心的代理设为AppDelegate:

```
UNUserNotificationCenter.current().delegate = self
```

再次运行app，设置一个推送测试一下。当推送出现时，点击稍后提醒我的按钮，则会在15分钟后收到另一个推送。

总结

希望这篇关于UserNotifications框架的介绍性教程对你有所帮助，在下一篇中我们会更深入的讨论如何在推送中嵌入自定义的视图控制器。

可以在Github上下载项目Demo。

Building a Barcode and QR Code Reader in Swift 3 and Xcode 8

Xcode 8中使用Swift 3构建条形码与QR码识别器

> - 原文链接：Building a Barcode and QR Code Reader in Swift 3 and Xcode 8
> - 原文作者：SIMON NG
> - 译者：yrq110



QR码是什么？我相信大部分人应该都知道QR码，如果没听说过可以看看上面那张图 - 那就是QR码。

QR(Quick Response的缩写)码是由Denso开发的一种二维条形码，最初是被设计用来跟踪制造过程的，近些年经常被用来对登录页面或商品信息的URL地址编码。跟普通的条形码不同的是，QR码包含水平与竖直两个方向的信息。在这里我不想讨论QR码的技术细节，若感兴趣可以看看QR码的官网。

随着iPhone与Android智能手机的流行，QR码的使用越来越广泛。在一些国家QR码随处可见，杂志上，报纸上，广告上，名片上，甚至是菜单上。作为一个iOS开发者，也许很想知道如何给你的app添加读取QR码的功能。在iOS 7之前需要依赖第三方库来实现识别功能，现在可以直接使用内置的AVFoundation框架进行实时识别条形码。

不过，做一个能够扫描与解析QR码的app一直不是个轻松的差事。

> 小提示: 可以在 http://www.qrcode-monkey.com 生成你自己的QR码。

构建QR码识别器

我们要做的demo是比较简单直接的，在开始构建之前需要了解一下iOS中的条形码与QR码扫描，它们都是基于视频捕捉的，这就是为何需要引入AVFoundation框架。记住这一点有助于对这一章节内容的理解。

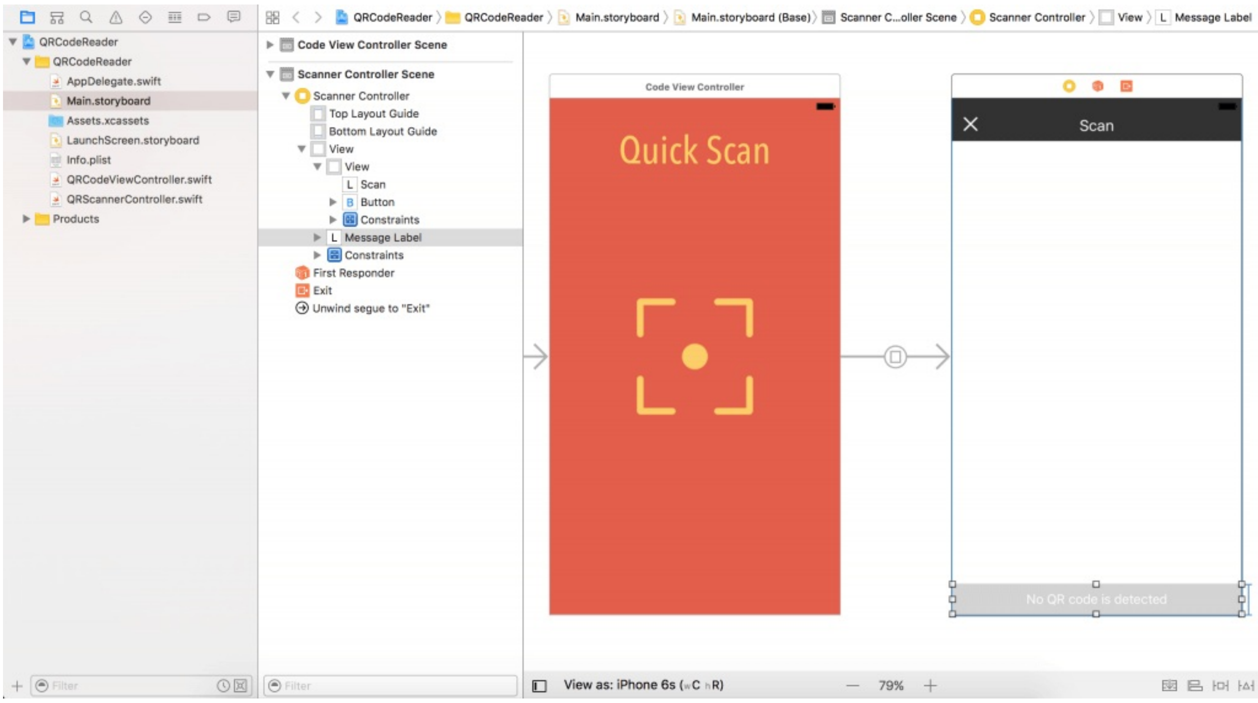
那么，demo是如何工作的?

看看下面的截图，这就是app界面的样子。app的功能有点像一个没有录像功能的视频捕捉app。当app启动后会利用iPhone的前置摄像头标示出QR码并自动识别，解码后的信息会显示在屏幕底部。



就是这么简单。

构建这个app前需要先下载项目模板，预先设置好了storyboard中的标签，主界面对应QRCodeViewController类，扫描界面对应QRScannerController类。



可以先运行一下试试，点击scan按钮会出现扫描视图，下面我们来实现这个视图中的QR码扫描功能。

导入AVFoundation框架

已经在项目模板中写好了app的UI界面，界面中的标签用来显示QR码解码后的信息，与QRScannerController类中的messageLabel属性关联。

之前强调过QR码的扫描功能是基于AVFoundation框架来实现的。首先打开QRScannerController.swift文件并导入AVFoundation框架:

```
import AVFoundation
```

接着需要实现AVCaptureMetadataOutputObjectsDelegate协议中的委托方法，这个后面再说，如下添加协议:

```
class ViewController: UIViewController, AVCaptureMetadataOutputObjectsDelegate
```

在进行下一步之前在QRScannerController类中声明以下变量:

```
var captureSession:AVCaptureSession?
var videoPreviewLayer:AVCaptureVideoPreviewLayer?
var qrCodeFrameView:UIView?
```

实现视频捕捉

QR码的获取是使用视频捕捉来实现的，为了进行实时的视频捕捉需要使用AVCaptureSession对象，设置其输入为用于视频捕捉的AVCaptureDevice。在QRScannerController类的viewDidLoad方法中插入如下代码:

```
// Get an instance of the AVCaptureDevice class to initialize a device object and prov
ide the video as the media type parameter.
let captureDevice = AVCaptureDevice.defaultDevice(withMediaType: AVMediaTypeVideo)

do {
    // Get an instance of the AVCaptureDeviceInput class using the previous device obj
ect.
    let input = try AVCaptureDeviceInput(device: captureDevice)

    // Initialize the captureSession object.
    captureSession = AVCaptureSession()

    // Set the input device on the capture session.
    captureSession?.addInput(input)

} catch {
    // If any error occurs, simply print it out and don't continue any more.
    print(error)
    return
}
```

AVCaptureDevice对象表示一个物理捕捉设备。为了获取视频数据，需要调用defaultDevice(withMediaType:)方法，输入AVMediaTypeVideo得到视频捕捉设备。

为了进行实时的视频捕捉需要初始化一个AVCaptureSession对象并添加视频捕捉设备作为输入，AVCaptureSession对象用来协调从视频输入到输出的数据流。

在这里，session的输出被设为一个AVCaptureMetaDataOutput对象。AVCaptureMetaDataOutput类是QR码识别中的核心部分，这个类结合AVCaptureMetadataOutputObjectsDelegate协议用来截获输入设备中的所有元数据(由设备摄像头所捕捉的QR码)，并将其转换成便于人们可读的格式。

没有理解的话别担心，一会儿就云开雾散了。在viewDidLoad方法中的do代码块中添加如下代码:

```
// Initialize a AVCaptureMetadataOutput object and set it as the output device to the
capture session.
let captureMetadataOutput = AVCaptureMetadataOutput()
captureSession?.addOutput(captureMetadataOutput)
```

接着添加下列代码，将captureMetadataOutput对象的委托设为自身(self)，这样就使QRReaderViewController实现了AVCaptureMetadataOutputObjectsDelegate协议。

```
// Set delegate and use the default dispatch queue to execute the call back
captureMetadataOutput.setMetadataObjectsDelegate(self, queue: DispatchQueue.main)
captureMetadataOutput.metadataObjectTypes = [AVMetadataObjectTypeQRCode]
```

当捕捉到原数据对象时会将其传给委托对象进行下一步的处理。在上面的代码中，指定了执行委托方法的调度队列(dispatch queue)，一个调度队列是串行或并行的。根据苹果的文档，这个队列必须是一个串行队列(serial queue)，因此需要使用DispatchQueue.main取得默认的串行队列。

metadataObjectTypes属性同样很重要，通过这个属性告诉app需要的是哪种格式的元数据，AVMetadataObjectTypeQRCode指明了我们的目标 —— QR码。

现在来设置AVCaptureMetadataOutput对象。使用AVCaptureVideoPreviewLayer在屏幕上显示设备摄像头捕捉到的视频，实际上是一个CALayer。结合预览层(preview layer)与AV capture session来显示视频，预览层作为当前视图的一个子层添加。将如下代码插入到do-catch代码块中:

```
// Initialize the video preview layer and add it as a sublayer to the viewPreview view
's layer.
videoPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
videoPreviewLayer?.videoGravity = AVLayerVideoGravityResizeAspectFill
videoPreviewLayer?.frame = view.layer.bounds
view.layer.addSublayer(videoPreviewLayer!)
```

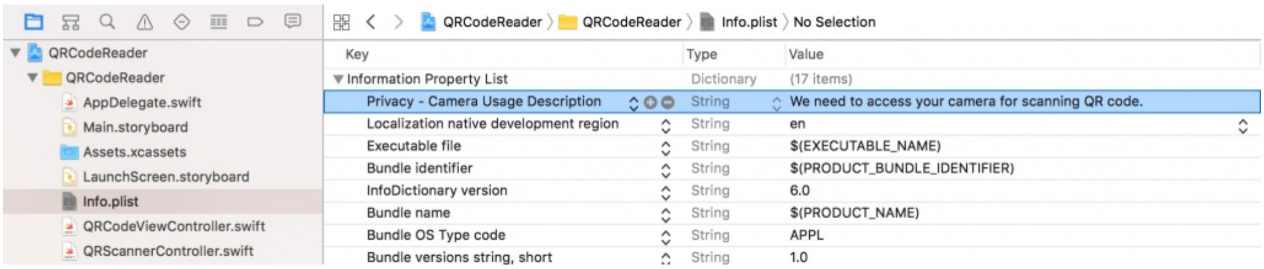
最后，调用capture session的startRunning方法启动视频捕捉:

```
// Start video capture.
captureSession?.startRunning()
```

若在iOS真机上编译运行app会崩溃并出现以下错误:

```
This app has crashed because it attempted to access privacy-sensitive data without a u
sage description.  The app's Info.plist must contain an NSCameraUsageDescription key w
ith a string value explaining to the user how the app uses this data.
```

iOS要求app开发者在访问摄像头之前需要取得用户的许可。解决方法是在Info.plist中添加一个NSCameraUsageDescription的key，打开文件后在空白区域点击右键选择添加新的一行，将key设为 `Privacy - Camera Usage Description` ，value设为 `We need to access your camera for scanning QR code` 。



完成编辑后，再次部署app并在真机上运行。点击scan按钮启动摄像头开始捕捉视频，会发现信息标签和顶部工具栏被隐藏了，可以添加如下代码来解决这个问题，它会将这两个组件移动到视频层的顶部。

```
// Move the message label and top bar to the front
view.bringSubview(toFront: messageLabel)
view.bringSubview(toFront: topbar)
```

修改完成后再次重启，屏幕中的信息标签会显示*No QR code is detected*。

实现读取QR码

现在这个app看起来像那么回事儿了，那么如何让它能在扫描QR码之后还能解析出其中的内容？其实app已经具有检测QR码这个功能了，只是还没有挖掘出来。如下修改一下app:

- 高亮检测到的QR码。
- 将解码后的信息显示在屏幕底端。

初始化高亮视图

为了高亮显示QR码需要创建一个绿色边界的UIView对象。在viewDidLoad方法的do代码块中添加如下代码:

```
// Initialize QR Code Frame to highlight the QR code
qrCodeFrameView = UIView()

if let qrCodeFrameView = qrCodeFrameView {
    qrCodeFrameView.layer.borderColor = UIColor.green.cgColor
    qrCodeFrameView.layer.borderWidth = 2
    view.addSubview(qrCodeFrameView)
    view.bringSubview(toFront: qrCodeFrameView)
}
```

qrCodeFrameView在屏幕中是不可见的，因为默认情况下它的尺寸是0，当检测到QR码时会设置它的尺寸并将其设为一个绿色边框的视图。

解析QR码

当AVCaptureMetadataOutput对象识别出QR码时，会调用AVCaptureMetadataOutputObjectsDelegate中的委托方法:

```
optional func captureOutput(_ captureOutput: AVCaptureOutput!, didOutputMetadataObjects metadataObjects: [Any]!, from connection: AVCaptureConnection!)
```

由于至此还未实现这个方法，因此无法解析QR码。为了捕捉并解析QR码中的信息，需要实现这个方法对元数据对象执行额外的操作。代码如下:

```
func captureOutput(_ captureOutput: AVCaptureOutput!, didOutputMetadataObjects metadataObjects: [Any]!, from connection: AVCaptureConnection!) {

    // Check if the metadataObjects array is not nil and it contains at least one object.
    if metadataObjects == nil || metadataObjects.count == 0 {
        qrCodeFrameView?.frame = CGRect.zero
        messageLabel.text = "No QR code is detected"
        return
    }

    // Get the metadata object.
    let metadataObj = metadataObjects[0] as! AVMetadataMachineReadableCodeObject

    if metadataObj.type == AVMetadataObjectTypeQRCode {
        // If the found metadata is equal to the QR code metadata then update the status label's text and set the bounds
        let barCodeObject = videoPreviewLayer?.transformedMetadataObject(for: metadataObj)
        qrCodeFrameView?.frame = barCodeObject!.bounds

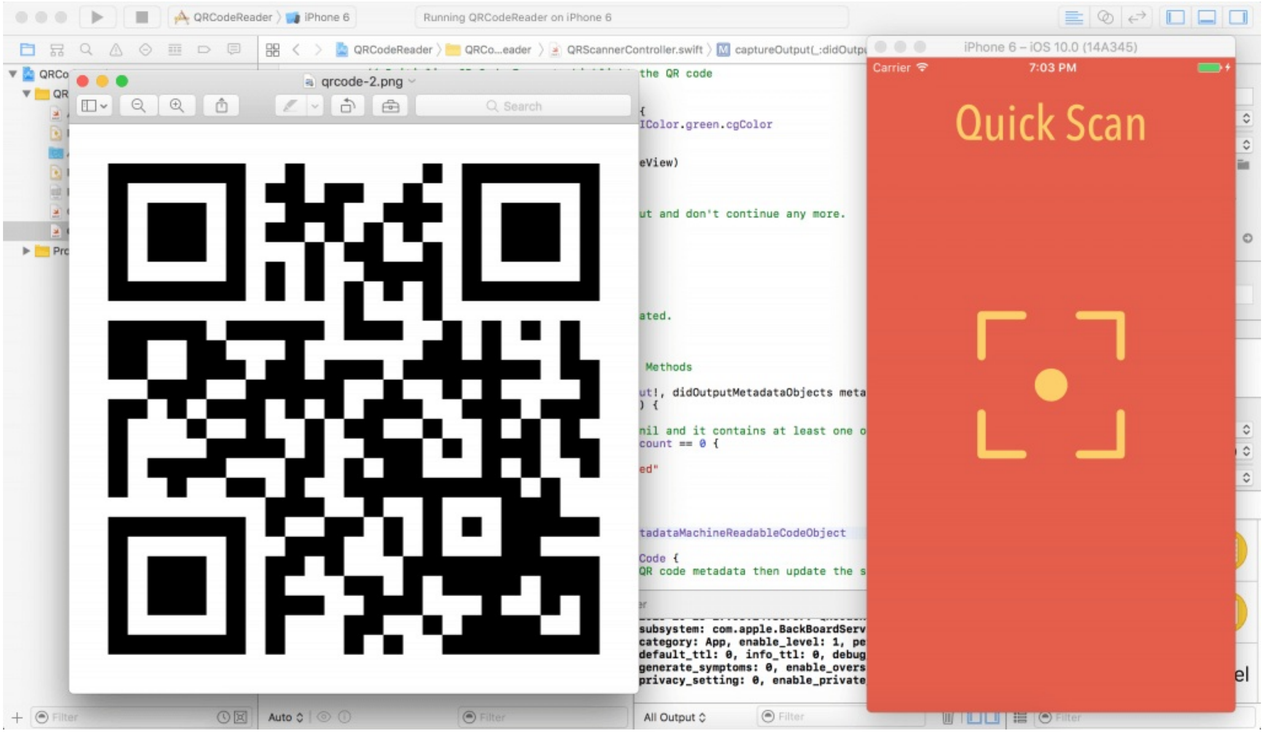
        if metadataObj.stringValue != nil {
            messageLabel.text = metadataObj.stringValue
        }
    }
}
```

方法的第二个参数(metadataObjects)是一个包含所有元数据对象的数组。首先要做的是确认这个数组是否非空，至少需要包含一个对象，否则需要将qrCodeFrameView的尺寸设为0并重置messageLabel所显示的信息。

若存在元数据对象，还需要验证是否是QR码，若是则需要找到QR码的边界，上面那些代码用来设置高亮QR码的视图。通过调用viewPreviewLayer的transformedMetadataObject(for:)方法获取元数据对象对应的视图层坐标，这样就可以确定所要添加的绿色边框视图的尺寸与位置了。

最后将QR码解析成人类可读的信息，这一步很简单，可以使用AVMetadataMachineReadableCode对象的stringValue属性访问解码出的信息。

准备好了！按下Run键在真机上编译并运行app。



启动后点击scan按钮将设备对准QR码，app会立刻检测到QR码并解码出其中的信息。



练习 – 条形码识别器

demo app现在可以识别出QR码，如果你能将它变成一个通用的条形码识别器是不是很棒？除了QR码，AVFoundation框架还支持如下类型的条形码:

- UPC-E (AVMetadataObjectTypeUPCECode)
- Code 39 (AVMetadataObjectTypeCode39Code)
- Code 39 mod 43 (AVMetadataObjectTypeCode39Mod43Code)
- Code 93 (AVMetadataObjectTypeCode93Code)
- Code 128 (AVMetadataObjectTypeCode128Code)
- EAN-8 (AVMetadataObjectTypeEAN8Code)
- EAN-13 (AVMetadataObjectTypeEAN13Code)
- Aztec (AVMetadataObjectTypeAztecCode)
- PDF417 (AVMetadataObjectTypePDF417Code)



你的任务就是修改Xcode项目使其能够扫描其他类型的条形码，需要设置captureMetadataOutput的metadataObjectTypes属性使其可以识别多种条形码。

```
captureMetadataOutput.metadataObjectTypes = [AVMetadataObjectTypeQRCode]
```

可以在Gihub下载完整代码。

Core Plot Tutorial: Getting Started

Core Plot指南:入门

> - 原文链接 : Core Plot Tutorial: Getting Started
> - 原文作者 : Attila Hegedüs
> - 译者 : yrq110

Core Plot是一个2D图表库，可以用于iOS、Mac OS X和tvOS，使用了苹果的Quartz与Core Animation框架实现，在BSD许可下发布。

在这篇教程中，你将会学到如何使用Core Plot创建饼图与条形图，还有如何添加很酷的图表交互功能！

开始前需要安装Xcode 8.0并且对Swift、Interface Builder和storyboard有一些基本的认识与理解，如果你对这些方面不太了解需要看看其他教程学习一下。

教程中使用CocoaPods来安装依赖的第三方库，若你没用过CocoaPods可以先看看这个教程学习一下。

入门

在这篇教程中会做一个显示货币汇率的app。在这里下载开始项目，解压后打开SwiftRates.xcworkspace。

项目中的一些关键类:

- `DataStore.swift` 一个从Fixer.io请求汇率数据的辅助类。
- `Rate.swift` 显示所选日期货币汇率的model。
- `Currency.swift` 货币类型的model，所支持的货币在Resources/Currencies.plist中定义。
- `MenuViewController.swift` app启动时显示的第一个view controller，在这里用户选择一个基本货币和两个比较货币。
- `HostViewController.swift` 一个view controller容器，根据分段控件的选择来显示PieChartViewController或BarGraphViewController，并且将从DataStore中获取的汇率数据应用于所选择的view controller。
- `PieChartViewController.swift` 使用饼图显示所选日期的汇率，首先会实现这个图表。

- `BarGraphViewController.swift` 使用条形图显示一定天数的汇率.实现饼图后这就是个小case！

构建并运行一下看看。



选择Get Rates进入HostViewController界面，改变分段控件的选择，app还没有实现这些功能...

是时候学习使用Core Plot绘图了!

安装Core Plot

首先需要安装Core Plot，使用CocoaPods安装比较简单。

将下面这行添加到Podfile中:

```
pod 'CorePlot', '~> 2.2'
```

打开终端，进入项目目录，执行pod install。

安装完成后构建项目。没毛病，现在准备好Core Plot了，谢谢你，CocoaPods :]

如果出了一些错误，可以尝试一下使用sudo gem install cocoapods更新CocoaPods然后再执行pod install。

创建饼图

打开PieChartViewController.swift文件导入Core Plot:

```
import CorePlot
```

接着添加如下属性:

```
@IBOutlet weak var hostView: CPTGraphHostingView!
```

CPTGraphHostingView负责"管理"一个图表(graph)，可以把它当做一个"图表容器"。

接着，在类的大括号结束符后添加如下类扩展:

```
extension PieChartViewController: CPTPieChartDataSource, CPTPieChartDelegate {

  func numberOfRecords(for plot: CPTPlot) -> UInt {
    return 0
  }

  func number(for plot: CPTPlot, field fieldEnum: UInt, record idx: UInt) -> Any? {
    return 0
  }

  func dataLabel(for plot: CPTPlot, record idx: UInt) -> CPTLayer? {
    return nil
  }

  func sliceFill(for pieChart: CPTPieChart, record idx: UInt) -> CPTFill? {
    return nil
  }

  func legendTitle(for pieChart: CPTPieChart, record idx: UInt) -> String? {
    return nil
  }
}
```

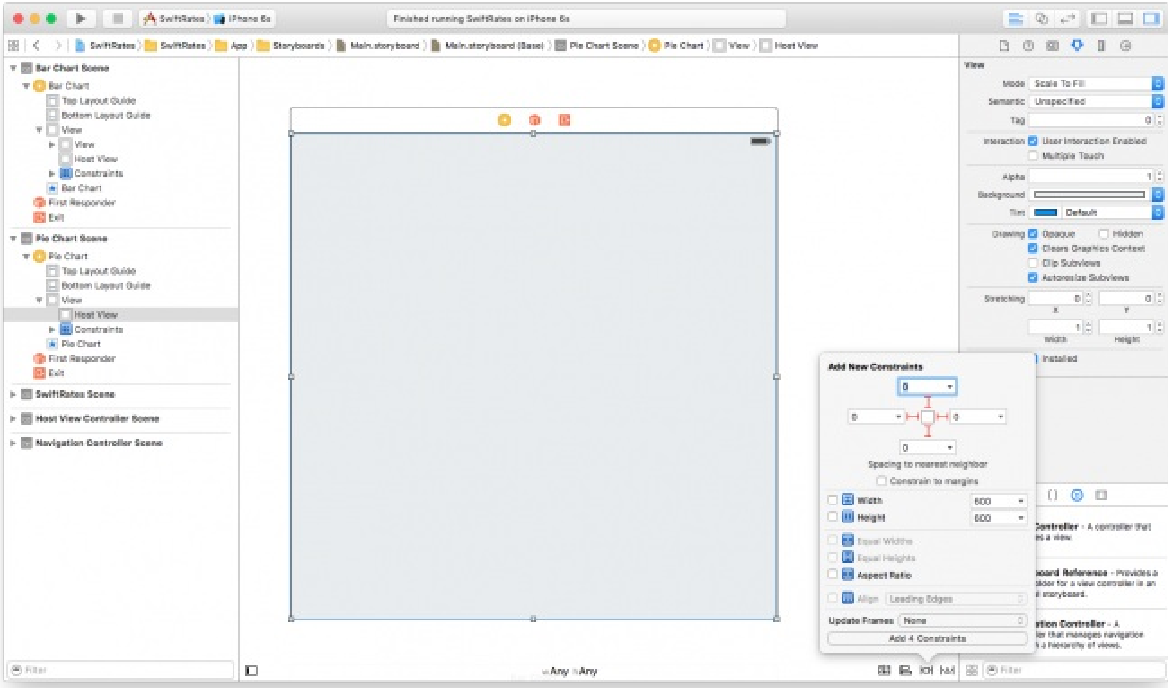
通过CPTPieChartDataSource提供Core Plot图表的数据，通过CPTPieChartDelegate得到用户的响应事件，会在之后填充这些方法的内容。

设置graph管理视图

打开Main.storyboard，选择PieChartViewController。

把一个新的UIView拖拽到这个视图中，将它的类改为CPTGraphHostingView，连接到hostView。

在每条边上添加约束使其嵌在父视图上，确保Constrain to margins没有被勾选:



背景色随意设置，我使用透明度是92%的灰度色。

回到PieChartViewController.swif文件，在viewDidLoad()后添加如下方法:

```
override func viewDidLayoutSubviews() {
  super.viewDidLayoutSubviews()
  initPlot()
}

func initPlot() {
  configureHostView()
  configureGraph()
  configureChart()
  configureLegend()
}

func configureHostView() {
}

func configureGraph() {
}

func configureChart() {
}

func configureLegend() {
}
```

子视图布局好后在这里设置需要绘制的图标，这里是最早设置视图尺寸的地方，需要在这里设置图表的参数。

initPlot()中的每一个方法都是图标设置中的一环，这种写法使代码更具组织性。

添加如下代码到configureHostView()方法中:

```
hostView.allowPinchScaling = false
```

这里禁用了饼图的捏合缩放，决定饼图的管理视图是否会对捏合手势作出响应。

接着在hostView中添加一个graph。在configureGraph()方法中添加如下代码:

```
// 1 - 创建并设置graph
let graph = CPTXYGraph(frame: hostView.bounds)
hostView.hostedGraph = graph
graph.paddingLeft = 0.0
graph.paddingTop = 0.0
graph.paddingRight = 0.0
graph.paddingBottom = 0.0
graph.axisSet = nil

// 2 - 创建文本样式
let textStyle: CPTMutableTextStyle = CPTMutableTextStyle()
textStyle.color = CPTColor.black()
textStyle.fontName = "HelveticaNeue-Bold"
textStyle.fontSize = 16.0
textStyle.textAlignment = .center

// 3 - 设置graph的标题与文本样式
graph.title = "\(base.name) exchange rates\n\(rate.date)"
graph.titleTextStyle = textStyle
graph.titlePlotAreaFrameAnchor = CPTRectAnchor.top
```

下面是对代码的分解:

1. 首先新建了一个CPTXYGraph实例，将其设为hostView的hostedGraph属性值，结合图形与管理视图。CPTGraph包含一个标准图表中的所有元素：边界、标题、绘制的数据、坐标轴和图例。默认情况下，CPTXYGraph的每一条边有20px的内边距。看起来并不是很好，你需要将它设为0。
2. 接着通过创建CPTMutableTextStyle实例设置图形的文本样式。
3. 最后设置图表的标题，将它的样式设为刚刚创建好的样式，并将锚点设为视图矩形的顶部。

构建并运行app，你应该会看到屏幕上方显示的graph标题。



绘制饼图

标题看着还行，不过怎么能做的更好？来看看如何绘制饼图！

将如下代码添加到configureChart()方法中:

```
// 1 - 获取graph引用
let graph = hostView.hostedGraph!

// 2 - 创建chart
let pieChart = CPTPieChart()
pieChart.delegate = self
pieChart.dataSource = self
pieChart.pieRadius = (min(hostView.bounds.size.width, hostView.bounds.size.height) * 0
.7) / 2
pieChart.identifier = NSString(string: graph.title!)
pieChart.startAngle = CGFloat(M_PI_4)
pieChart.sliceDirection = .clockwise
pieChart.labelOffset = -0.6 * pieChart.pieRadius

// 3 - 设置边界样式
let borderStyle = CPTMutableLineStyle()
borderStyle.lineColor = CPTColor.white()
borderStyle.lineWidth = 2.0
pieChart.borderLineStyle = borderStyle

// 4 - 设置文本样式
let textStyle = CPTMutableTextStyle()
textStyle.color = CPTColor.white()
textStyle.textAlignment = .center
pieChart.labelTextStyle = textStyle

// 5 - 将chart添加到graph中
graph.add(pieChart)
```

这些代码干了什么:

1. 首先得到一个图的引用。
2. 接着实例化CPTPieChart，设置它的委托与数据源为view controller，并设置它的外观。
3. 设置表的边界样式。
4. 设置表的文本样式。
5. 最后将chart添加到graph中。

如果你现在就运行app的话会看不到任何改变...这是因为还需要为饼图实现数据源与委托。

首先使用如下代码替换numberOfRecords(for:)中的内容:

```
func numberOfRecords(for plot: CPTPlot) -> UInt {
  return UInt(symbols.count)
}
```

这个方法决定了在图中显示的分块数，会为每一种货币分配一个扇形区域。

接着使用如下代码替换number(for:field:record:)中的内容:

```
func number(for plot: CPTPlot, field fieldEnum: UInt, record idx: UInt) -> Any? {
  let symbol = symbols[Int(idx)]
  let currencyRate = rate.rates[symbol.name]!.floatValue
  return 1.0 / currencyRate
}
```

饼图使用这个方法在idx中得到货币符号的值。

应该注意到了这个值并不是个百分率，这个方法计算当前货币相对于基本货币的汇率：返回值1.0 / currencyRate是“当前货币相对于1个基本货币”的值。

CPTPieChart会计算每一块区域的百分比值，使用这些值决定扇形区域的大小。

接着，使用如下代码替换dataLabelForPlot(for:record:)方法:

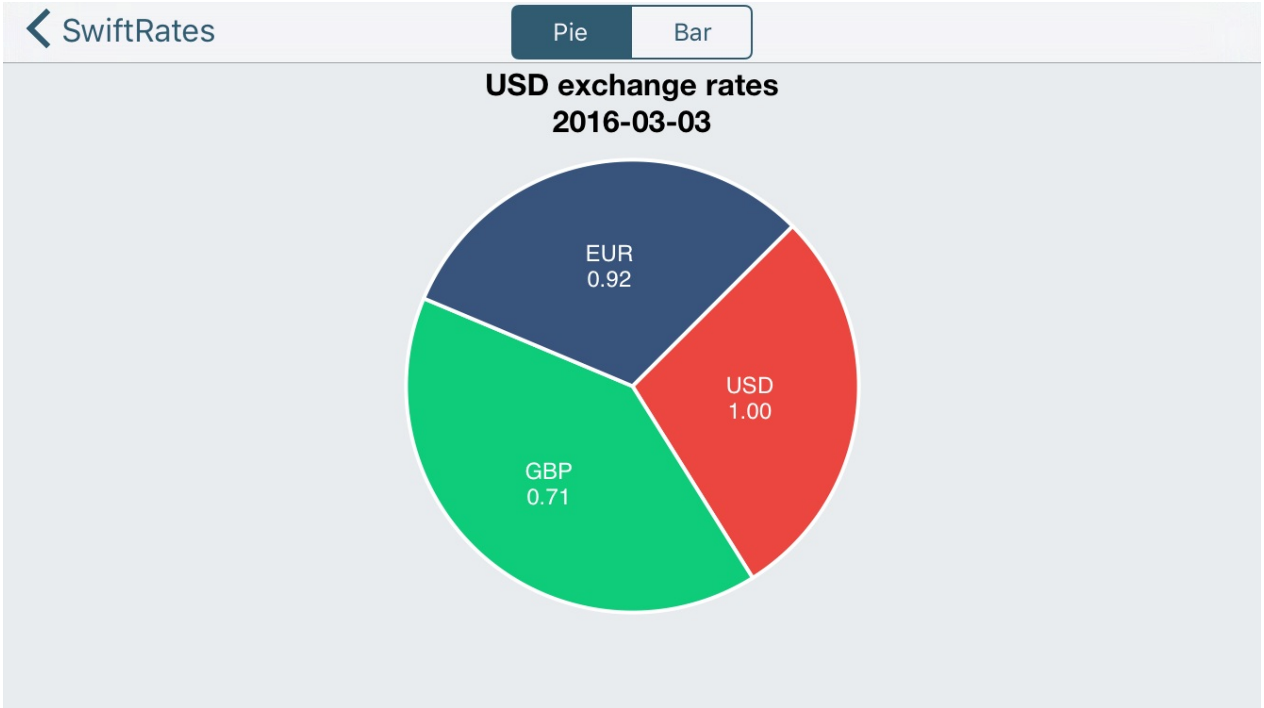
```
func dataLabel(for plot: CPTPlot, record idx: UInt) -> CPTLayer? {
  let value = rate.rates[symbols[Int(idx)].name]!.floatValue
  let layer = CPTTextLayer(text: String(format: "\(symbols[Int(idx)].name)\n%.2f", val
ue))
  layer.textStyle = plot.labelTextStyle
  return layer
}
```

这个方法返回饼图区域的标签，返回的类型是CPTLayer，它与CALayer很像，不同的是CPTLayer可以在macOS和iOS上使用，并且提供了Core Plot的一些绘制细节。

在这里新建并返回了一个CPTTextLayer对象，它用于显示文本的CPTLayer子类。最后给饼图中的区域赋予颜色。使用如下代码替换sliceFillForPieChart(for:, record:)方法:

```
func sliceFill(for pieChart: CPTPieChart, record idx: UInt) -> CPTFill? {
  switch idx {
  case 0:   return CPTFill(color: CPTColor(componentRed:0.92, green:0.28, blue:0.25, a
lpha:1.00))
  case 1:   return CPTFill(color: CPTColor(componentRed:0.06, green:0.80, blue:0.48, a
lpha:1.00))
  case 2:   return CPTFill(color: CPTColor(componentRed:0.22, green:0.33, blue:0.49, a
lpha:1.00))
  default:  return nil
  }
}
```

构建并运行，会看到一个漂亮的饼图:



加些图注!

图表看起来不错，不过有图注的话会更好，接下来给图表添加一个图注。

首先，使用如下代码替换configureLegend()方法中的内容:

```
func configureLegend() {
  // 1 - 获取graph对象
  guard let graph = hostView.hostedGraph else { return }

  // 2 - 创建图注
  let theLegend = CPTLegend(graph: graph)

  // 3 - 设置图注
  theLegend.numberOfColumns = 1
  theLegend.fill = CPTFill(color: CPTColor.white())
  let textStyle = CPTMutableTextStyle()
  textStyle.fontSize = 18
  theLegend.textStyle = textStyle

  // 4 - 将图注添加到图
  graph.legend = theLegend
  if view.bounds.width > view.bounds.height {
    graph.legendAnchor = .right
    graph.legendDisplacement = CGPoint(x: -20, y: 0.0)

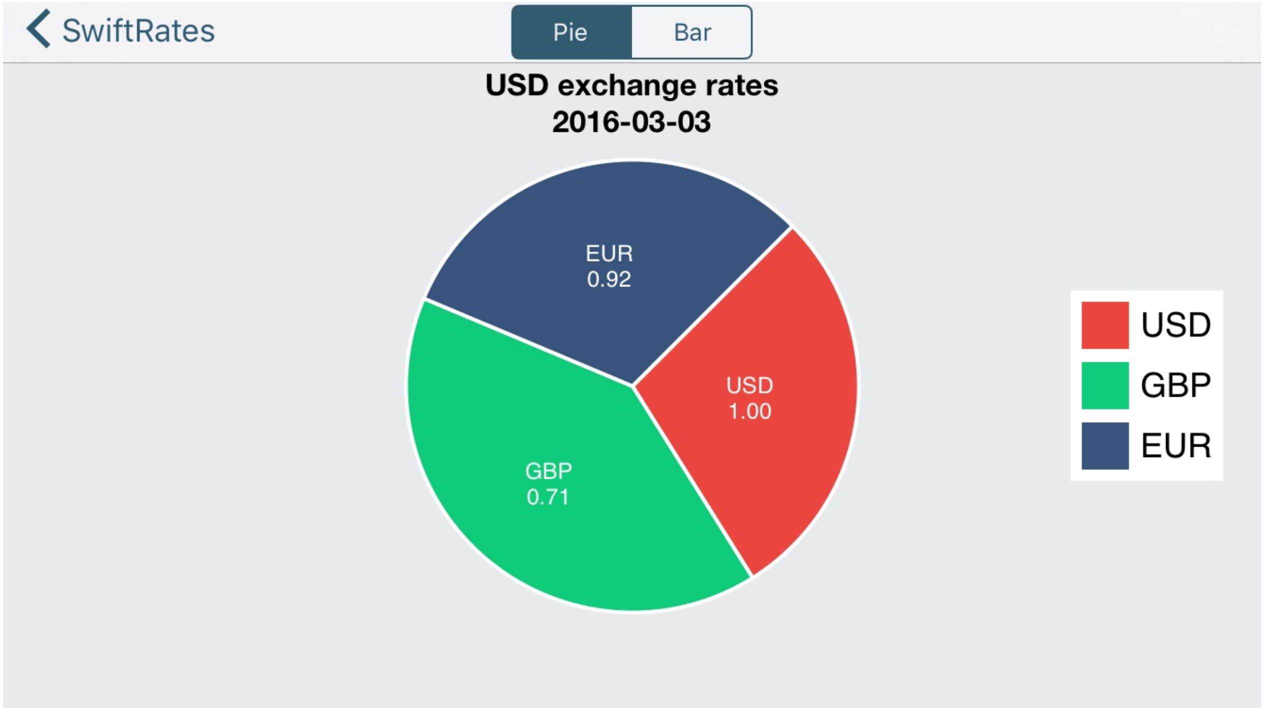
  } else {
    graph.legendAnchor = .bottomRight
    graph.legendDisplacement = CGPoint(x: -8.0, y: 8.0)
  }
}
```

还需要提供每一片区域的图注数据。

使用如下代码替换legendTitleForPieChart(for:record:)方法中的内容:

```
func legendTitle(for pieChart: CPTPieChart, record idx: UInt) -> String? {
  return symbols[Int(idx)].name
}
```

构建并运行，迎接一个"传奇(legendary，包含图注的，这里一语双关)"的图表。



创建条形图(Raising the Bar，提高标准，这里双关)

现在你画饼图应该挺6了，是时候提高一些标准，试下画条形图了!

打开BarGraphViewController，导入Core Plot:

```
import CorePlot
```

添加一个outlet:

```
@IBOutlet var hostView: CPTGraphHostingView!
```

像画饼图一样，host view需要包含这个条形图。

添加如下属性:

```
var plot1: CPTBarPlot!
var plot2: CPTBarPlot!
var plot3: CPTBarPlot!
```

这里创建的三个CPTBarPlot属性分别对应graph中的每一种货币。

注意这里已经定义了三个IBOutlet标签与三个IBAction方法，并且已与故事板连接。

最后，在文件末尾处添加如下扩展:

```
extension BarGraphViewController: CPTBarPlotDataSource, CPTBarPlotDelegate {

  func numberOfRecords(for plot: CPTPlot) -> UInt {
    return 0
  }

  func number(for plot: CPTPlot, field fieldEnum: UInt, record idx: UInt) -> Any? {
    return 0
  }

  func barPlot(_ plot: CPTBarPlot, barWasSelectedAtRecord idx: UInt, with event: UIEve
nt) {

  }
}
```

这里跟饼图很像: 通过CPTBarPlotDataSource给条形图提供数据，通过CPTBarPlotDelegate获取用户交互事件。

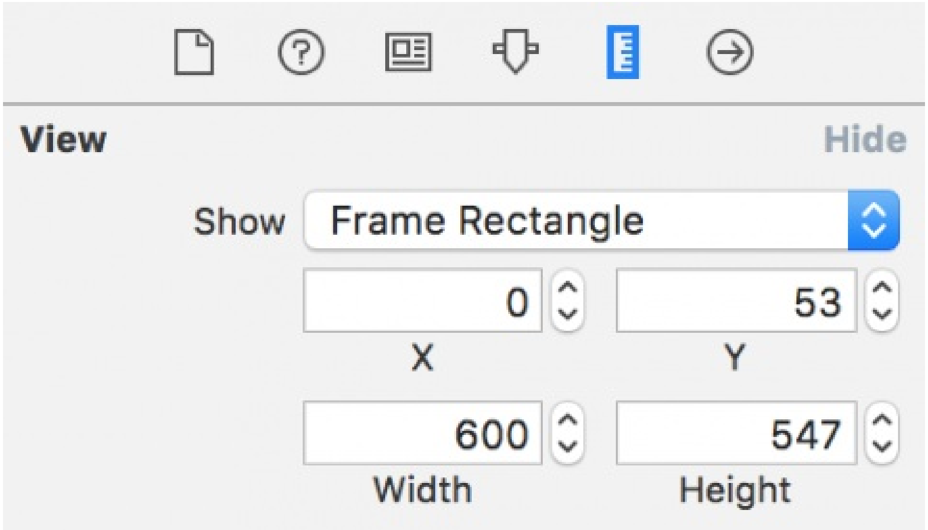
再次设置graph管理视图

跟刚才所做的一样，使用Interface Builder添加管理视图。

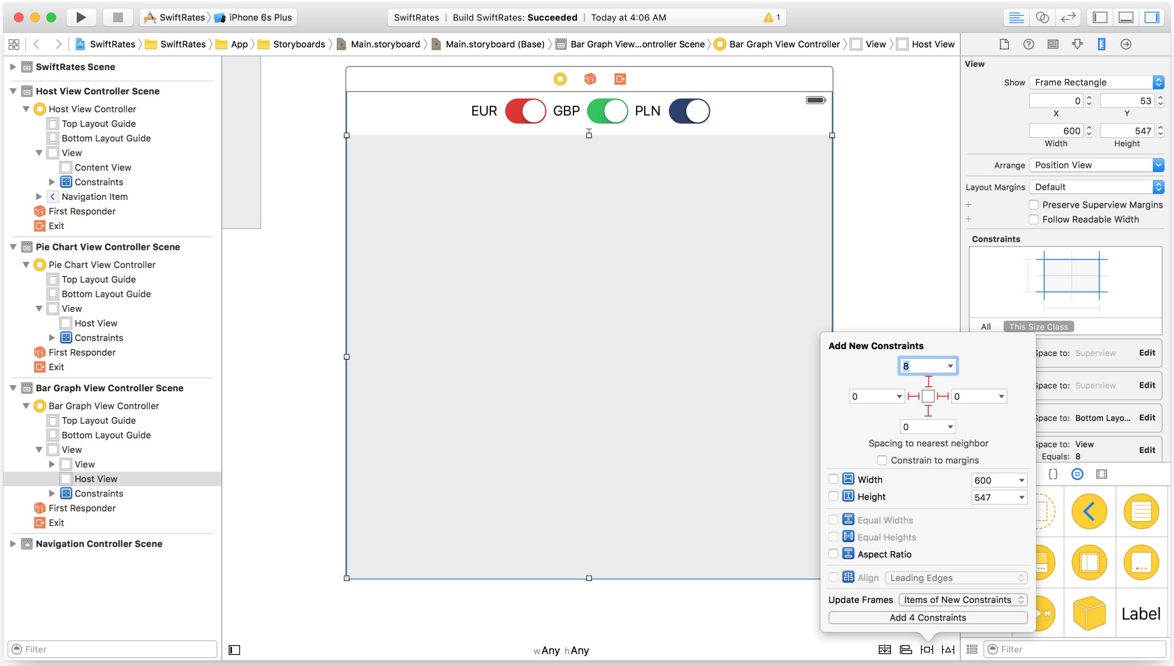
回到Main.storyboard中，选择BarGraphViewController场景。

把一个新的UIView拖到视图中，将class改为CPTGraphHostingView，将outlet连接到view controller的hostView对象上。

在Utilities\Size Inspector(尺子状的标签)中调整成如下尺寸: X = 0, Y = 53, Width = 600, Height = 547



添加四条边的约束，确保Constrain to margins没有被勾选。



最后设置任意的背景色，仍然使用的是透明度为92%的灰度色。

绘制条形图

UI组件都弄好了，现在来绘制条形图吧。

首先回到BarGraphViewController中，需要添加几个常量属性。在其他属性后添加如下属性:

```
let BarWidth = 0.25
let BarInitialX = 0.25
```

来需要一个辅助函数来计算最高汇率，在updateLabels()方法后添加如下函数:

```
func highestRateValue() -> Double {
  var maxRate = DBL_MIN
  for rate in rates {
    maxRate = max(maxRate, rate.maxRate().doubleValue)
  }
  return maxRate
}
```

接着在highestRateValue()后添加如下方法:

```
override func viewDidLayoutSubviews() {
  super.viewDidLayoutSubviews()
  initPlot()
}

func initPlot() {
  configureHostView()
  configureGraph()
  configureChart()
  configureAxes()
}

func configureHostView() {
}

func configureGraph() {
}

func configureChart() {
}

func configureAxes() {
}
```

是不是有点眼熟? 对的，这个结构跟之前一样。

在configureHostView()方法中添加如下:

```
hostView.allowPinchScaling = false
```

这次也不需要捏合缩放，禁用它。

在configureGraph()方法中添加如下代码:

```
// 1 - 新建graph
let graph = CPTXYGraph(frame: hostView.bounds)
graph.plotAreaFrame?.masksToBorder = false
hostView.hostedGraph = graph

// 2 - 设置graph
graph.apply(CPTTheme(named: CPTThemeName.plainWhiteTheme))
graph.fill = CPTFill(color: CPTColor.clear())
graph.paddingBottom = 30.0
graph.paddingLeft = 30.0
graph.paddingTop = 0.0
graph.paddingRight = 0.0

// 3 - 设置样式
let titleStyle = CPTMutableTextStyle()
titleStyle.color = CPTColor.black()
titleStyle.fontName = "HelveticaNeue-Bold"
titleStyle.fontSize = 16.0
titleStyle.textAlignment = .center
graph.titleTextStyle = titleStyle

let title = "\(base.name) exchange rates\n\(rates.first!.date) - \(rates.last!.date)"
graph.title = title
graph.titlePlotAreaFrameAnchor = .top
graph.titleDisplacement = CGPoint(x: 0.0, y: -16.0)

// 4 - 设置绘制区域
let xMin = 0.0
let xMax = Double(rates.count)
let yMin = 0.0
let yMax = 1.4 * highestRateValue()
guard let plotSpace = graph.defaultPlotSpace as? CPTXYPlotSpace else { return }
plotSpace.xRange = CPTPlotRange(locationDecimal: CPTDecimalFromDouble(xMin), lengthDec
imal: CPTDecimalFromDouble(xMax - xMin))
plotSpace.yRange = CPTPlotRange(locationDecimal: CPTDecimalFromDouble(yMin), lengthDec
imal: CPTDecimalFromDouble(yMax - yMin))
```

分析下上面的代码:

1. 首先实例化一个CPTXYGraph对象，其实就是一个条形图，将其与hostView连接起来。
2. 接着声明默认的主题为空白主题，设置上侧与左侧的内边距给坐标轴留出空间。
3. 设置文本样式，表格标题与标题位置。
4. 最后设置CPTXYPlotSpace，它负责将设备的坐标系映射到graph的坐标系。

在这个graph中需要在同一块区域内绘制出三种汇率，当然，也可以分别在不同区域进行绘制。

在绘图区域中会假定一个汇率的区间范围，在之后会看到如何在事先不知道这个范围的情况下让绘图区域的尺寸自动调节。

graph搞定后，是时候添加一些条形图了，在configureChart()方法中添加如下代码:

```
// 1 - 设置三个条形图
plot1 = CPTBarPlot()
plot1.fill = CPTFill(color: CPTColor(componentRed:0.92, green:0.28, blue:0.25, alpha:1
.00))
plot2 = CPTBarPlot()
plot2.fill = CPTFill(color: CPTColor(componentRed:0.06, green:0.80, blue:0.48, alpha:1
.00))
plot3 = CPTBarPlot()
plot3.fill = CPTFill(color: CPTColor(componentRed:0.22, green:0.33, blue:0.49, alpha:1
.00))

// 2 - 设置线条样式
let barLineStyle = CPTMutableLineStyle()
barLineStyle.lineColor = CPTColor.lightGray()
barLineStyle.lineWidth = 0.5

// 3 - 将条形添加到graph
guard let graph = hostView.hostedGraph else { return }
var barX = BarInitialX
let plots = [plot1!, plot2!, plot3!]
for plot: CPTBarPlot in plots {
  plot.dataSource = self
  plot.delegate = self
  plot.barWidth = NSNumber(value: BarWidth)
  plot.barOffset = NSNumber(value: barX)
  plot.lineStyle = barLineStyle
  graph.add(plot, to: graph.defaultPlotSpace)
  barX += BarWidth
}
```

上面的代码做了什么:

1. 实例化每个条形图并设置填充颜色。
2. 实例化一个CPTMutableLineStyle对象，它表示每个条形图外边界的样式。
3. 给每个条形图进行一个公共的配置，包含数据源与委托、每个图的宽度与相对距离、线条样式，最后将图形添加到graph中。

现在还看不到条形图，先构建一下app确保所有代码都能编译成功。

为了在条形图中显示数据，需要实现委托方法来给graph提供必要的数据。

使用如下代码替换numberOfRecordsForPlot(for:)中的内容:

```
return UInt(rates.count)
```

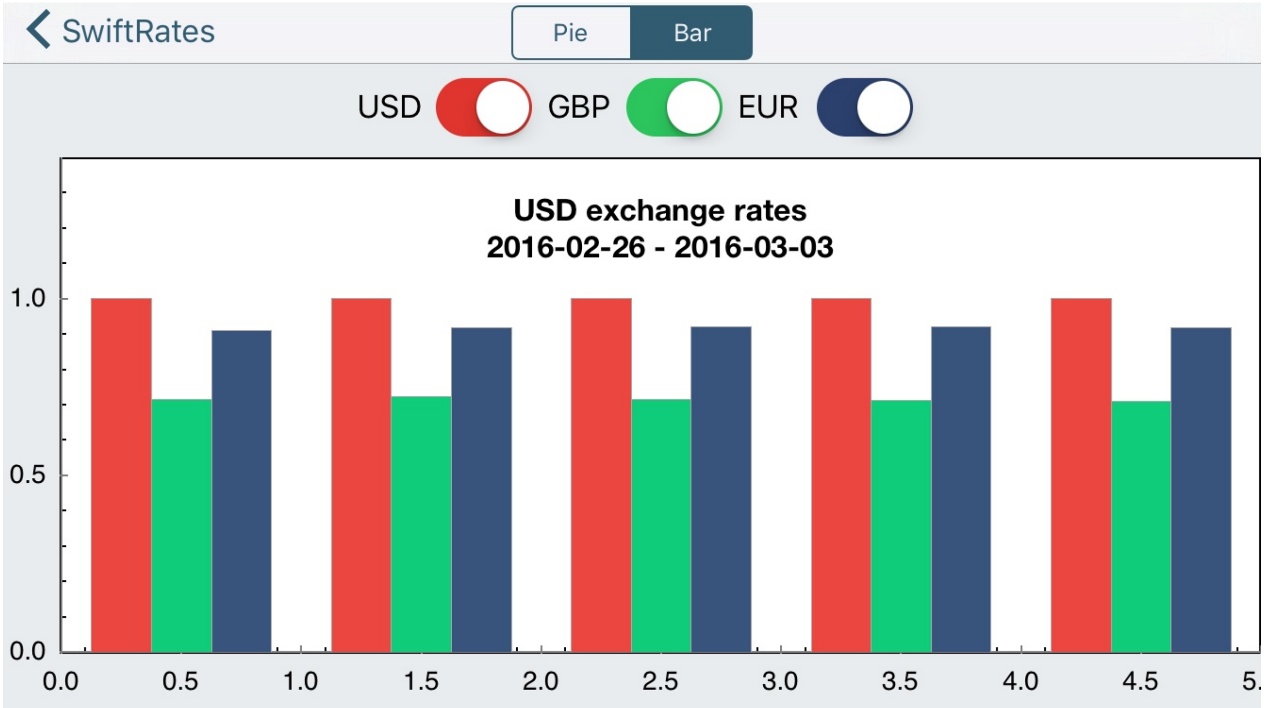
这个方法会返回要显示的记录个数。

使用如下代码替换numberForPlot(for:field:record:)中的内容:

```
if fieldEnum == UInt(CPTBarPlotField.barTip.rawValue) {
  if plot == plot1 {
    return 1.0 as AnyObject?
  }
  if plot == plot2 {
    return rates[Int(idx)].rates[symbols[0].name]!
  }
  if plot == plot3 {
    return rates[Int(idx)].rates[symbols[1].name]!
  }
}
return idx
```

CPTBarPlotField.BarTip属性表示条形图的相对尺寸，使用已有的属性进行检索得到所需的汇率数据，idx值为汇率的索引值。

构建并运行，应该会看到下面这样:



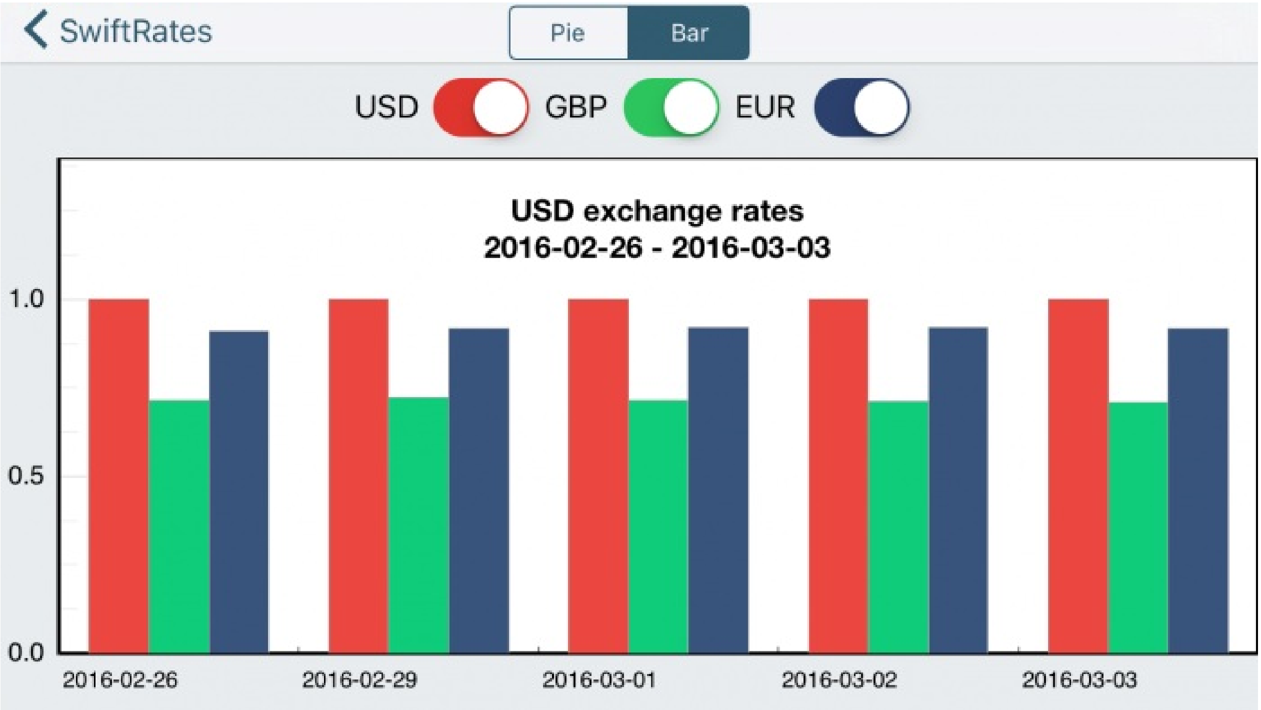
快成功了! 还有件事没做，你会注意到图中并没有坐标轴的标识信息。

在configureAxes()方法中添加如下代码来解决这个问题:

```
// 1 - 设置样式
let axisLineStyle = CPTMutableLineStyle()
axisLineStyle.lineWidth = 2.0
axisLineStyle.lineColor = CPTColor.black()
// 2 - 获取graph的轴线集
guard let axisSet = hostView.hostedGraph?.axisSet as? CPTXYAxisSet else { return }
// 3 - 设置x轴
if let xAxis = axisSet.xAxis {
  xAxis.labelingPolicy = .none
  xAxis.majorIntervalLength = 1
  xAxis.axisLineStyle = axisLineStyle
  var majorTickLocations = Set<NSNumber>()
  var axisLabels = Set<CPTAxisLabel>()
  for (idx, rate) in rates.enumerated() {
    majorTickLocations.insert(NSNumber(value: idx))
    let label = CPTAxisLabel(text: "\(rate.date)", textStyle: CPTTextStyle())
    label.tickLocation = NSNumber(value: idx)
    label.offset = 5.0
    label.alignment = .left
    axisLabels.insert(label)
  }
  xAxis.majorTickLocations = majorTickLocations
  xAxis.axisLabels = axisLabels
}
// 4 - 设置y轴
if let yAxis = axisSet.yAxis {
  yAxis.labelingPolicy = .fixedInterval
  yAxis.labelOffset = -10.0
  yAxis.minorTicksPerInterval = 3
  yAxis.majorTickLength = 30
  let majorTickLineStyle = CPTMutableLineStyle()
  majorTickLineStyle.lineColor = CPTColor.black().withAlphaComponent(0.1)
  yAxis.majorTickLineStyle = majorTickLineStyle
  yAxis.minorTickLength = 20
  let minorTickLineStyle = CPTMutableLineStyle()
  minorTickLineStyle.lineColor = CPTColor.black().withAlphaComponent(0.05)
  yAxis.minorTickLineStyle = minorTickLineStyle
  yAxis.axisLineStyle = axisLineStyle
}
```

简单的说，上面的代码首先定义了坐标轴线条与标签的样式，接着通过检索得到graph中的坐标轴集合，设置x轴与y轴。

构建并运行一下看看变化。



完善

的确更好，不是吗? 唯一的缺点就是坐标轴信息是空白的————没有给予任何关于汇率的准确信息。

可以这样来弥补这个问题：当用户点击每个条形图中的粗线条时显示具体的数据，为此需要增加一个属性:

```
var priceAnnotation: CPTPlotSpaceAnnotation?
```

接下来在barPlot(for:barWasSelectedAtRecord:with:)方法中添加如下代码:

```
// 1 - 条形图是否隐藏?
  if plot.isHidden == true {
    return
  }
  // 2 - 若未隐藏则创建文本样式
  let style = CPTMutableTextStyle()
  style.fontSize = 12.0
  style.fontName = "HelveticaNeue-Bold"

  // 3 - 创建标记
  guard let price = number(for: plot,
                           field: UInt(CPTBarPlotField.barTip.rawValue),
                           record: idx) as? CGFloat else { return }

  priceAnnotation?.annotationHostLayer?.removeAnnotation(priceAnnotation)
  priceAnnotation = CPTPlotSpaceAnnotation(plotSpace: plot.plotSpace!, anchorPlotPoint
: [0,0])

  // 4 - 创建数字格式化器
  let formatter = NumberFormatter()
  formatter.maximumFractionDigits = 2
  // 5 - 创建标记的文本层
  let priceValue = formatter.string(from: NSNumber(cgFloat: price))
  let textLayer = CPTTextLayer(text: priceValue, style: style)

  priceAnnotation!.contentLayer = textLayer
  // 6 - 获取条形图索引
  var plotIndex: Int = 0
  if plot == plot1 {
    plotIndex = 0
  }
  else if plot == plot2 {
    plotIndex = 1
  }
  else if plot == plot3 {
    plotIndex = 2
  }
  // 7 - 设置标记的锚点
  priceAnnotation!.anchorPlotPoint = [NSNumber(cgFloat: x), NSNumber(cgFloat: y)]
  // 8 - 添加标记
  guard let plotArea = plot.graph?.plotAreaFrame?.plotArea else { return }
  plotArea.addAnnotation(priceAnnotation)
}
```

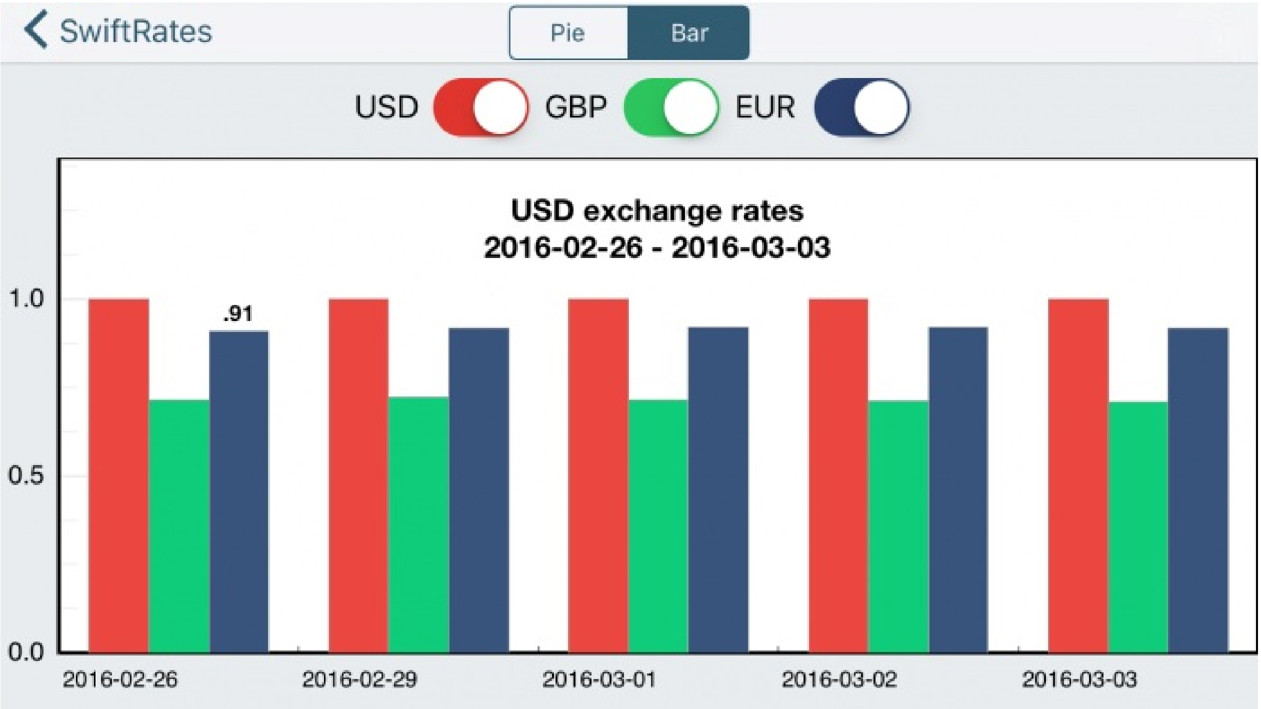
分析一下代码:

1. 如果条形图隐藏了则不会添加标记，直接返回，虽然条形图现在不具有隐藏的功能，会在下一部分来实现。
2. 创建标记的文本样式。
3. 得到所指定条形图的数值，并创建一个标记对象。

4. 创建一个数字格式化器，用它来格式化要显示的数据。
5. 创建一个文本层显示格式化后的数据，并将标记的内容层设置为这个文本层。
6. 获取需要显示标记的条形图索引。
7. 根据条形图的索引计算标记的位置，使用计算后的位置设置标记的锚点。
8. 最后将标记添加到graph中。

构建并运行，每当你点击条形图中的粗线条时，就会在它的上面显示出数值。

好极了! :]



隐藏和显现

条形图看起来不错，不过顶部的开关按钮貌似没什么作用，是时候实现它的功能了。

首先需要添加一个辅助方法，在switch3Changed(_:)后添加这个方法:

```
func hideAnnotation(graph: CPTGraph) {
  guard let plotArea = graph.plotAreaFrame?.plotArea,
    let priceAnnotation = priceAnnotation else {
      return
  }
  plotArea.removeAnnotation(priceAnnotation)
  self.priceAnnotation = nil
}
```

这段代码的作用是若存在标记则移除它。

下面想让用户使用开关按钮来改变条形图中所显示的货币。

为了实现它，需要使用下面的代码替换switch1Changed(_:), switch2Changed(_:)和switch3Changed(_:)方法:

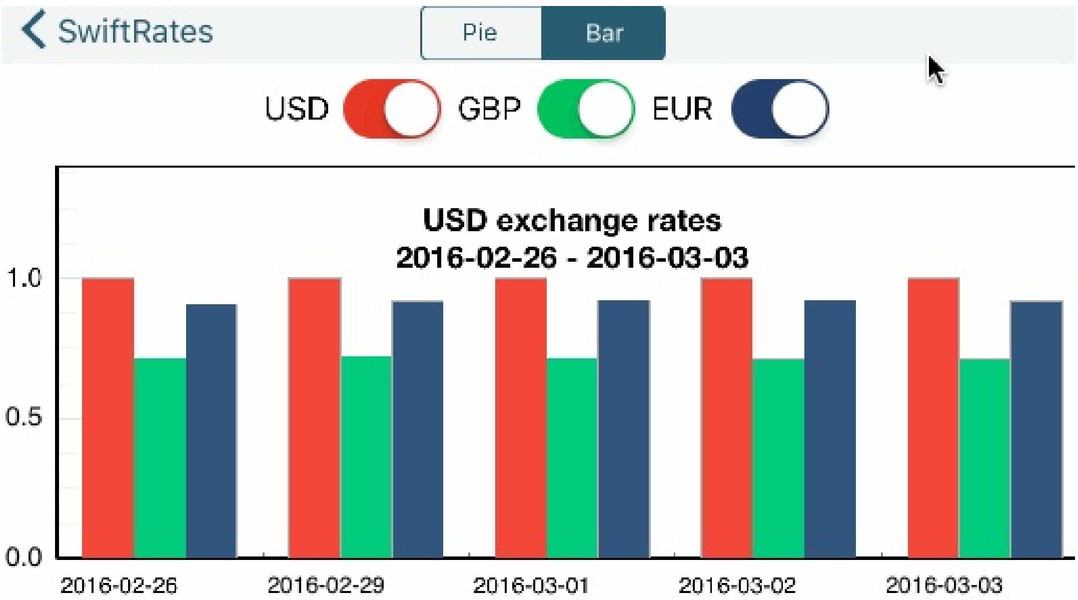
```
@IBAction func switch1Changed(_ sender: UISwitch) {
  let on = sender.isOn
  if !on {
    hideAnnotation(graph: plot1.graph!)
  }
  plot1.isHidden = !on
}

@IBAction func switch2Changed(_ sender: UISwitch) {
  let on = sender.isOn
  if !on {
    hideAnnotation(graph: plot2.graph!)
  }
  plot2.isHidden = !on
}

@IBAction func switch3Changed(_ sender: UISwitch) {
  let on = sender.isOn
  if !on {
    hideAnnotation(graph: plot3.graph!)
  }
  plot3.isHidden = !on
}
```

逻辑非常简单，如果开关按钮是关闭状态则隐藏对应的条形图和标记，若为打开状态则设置条形图为可见的。

构建并运行，可以随意切换显示的条形图，干的不错!



最后

可以从这里下载完整的项目文件。

Magical Error Handling in Swift

奇妙的Swift错误处理

> - 原文链接：Magical Error Handling in Swift
> - 原文作者：Gemma Barlow
> - 译者：yrq110



Swift的错误处理是从Swift 1借鉴OC的模式开始，在那之后取得了很大的进展，在Swift 2中的更新使得app中异常状态和场景的处理更加方便，这个优点延续到了Swift 3，不过在最新版本中却没有什么重大的更新。

像其它变成语言一样，swift中的错误处理方法会根据所遇到的错误类型与app的整体结构有所不同。

这篇教程会将你带入一个有巫师、魔女、蝙蝠和蟾蜍的魔法世界中，展示如何较好处理一些常见的错误场景，同时也会学到如何将项目中老版本的错误处理方法更新成新版的语法，最后，通过注视你的水晶球展望Swift错误处理的未来!

> 注意: 这篇教程假定你对Swift 3的语法较为熟悉 - 特别是枚举和可选类型。若需要重温这些概念，可以先看看Greg Heo的What's New in Swift 2或者其它资料。

是时候开始了(from the the cauldron into the fire!)，一同去发现Swift 3中错误处理的奇妙之处!

入门

这篇教程有两个开始项目，分别对应两个小节，使用nil处理错误和自定义错误处理方法。

在Xcode中打开第一个项目。

浏览一下代码会看到一些类、结构体与枚举，它们在这篇教程中具有魔力。

注意这部分代码:

```
protocol Avatar {
  var avatar: String { get }
}
```

这个协议基本上被所有类和结构体继承了，为每个对象提供一个可视化的表现属性使其可以输出到控制台中。

```
enum MagicWords: String {
  case abracadbra = "abracadabra"
  case alakazam = "alakazam"
  case hocusPocus = "hocus pocus"
  case prestoChango = "presto chango"
}
```

这个枚举表示用来创建魔法的咒语。

```
struct Spell {
  var magicWords: MagicWords = .abracadabra
}
```

用于构建一个基本的咒语，默认的咒语是.abracadabra。

好了现在你已经熟悉了这个超自然的世界，作为五火球神教的一员已经准备好施放魔法了。

为何重视错误处理?

> "错误处理是一门让错误变得优雅的艺术"–Swift Apprentice, 第22章 (错误处理)

良好的错误处理可以增强终端用户的体验，使软件维护者可以更容易的发现问题、问题起因和引起的严重性。对代码中的错误处理针对性越强，代码就越容易维护。同时错误处理可以在系统层面以合适的方式处理异常，以免给用户带来糟糕的体验。

不过错误并不总是需要被处理的，有时语言的特性就会避免一些常见的错误。比如说，若有些语法特性可以完全避免一个错误的发生可以直接使用这种设计，若无法避免一个潜在的错误条件(error condition)，针对性的处理是最好的选择。

使用nil避免错误

由于Swift优秀的可选处理能力，可以完全避免当期望一个值时但没有值存在时的错误条件。作为一个机智的开发者，可以利用这个特性在错误条件发生时返回nil，当遇到一个错误状态时应该不采取任何操作，这么做效果很好。

使用nil避免swift错误的两个典型示例是可失败构造器和guard语句。

可失败构造器

可失败构造器会阻止一个没有提供有效信息的对象的创建，在Swift提供这个构造器之前一般是用工厂模式来实现的。

在create:方法中可以看到这个模式的实现:

```
static func create(withMagicWords words: String) -> Spell? {
  if let incantation = MagicWords(rawValue: words) {
    var spell = Spell()
    spell.magicWords = incantation
    return spell
  }
  else {
    return nil
  }
}
```

上面的构造器尝试使用所提供的咒语来创建符咒，若咒语不是魔法咒语则返回nil。

观察一下创建符咒后的输出:

```
let first = Spell.create(withMagicWords: "abracadabra")                Spell
let second = Spell.create(withMagicWords: "ascendio")                  nil
```

第一次使用咒语"abracadabra"成功创建了符咒，不过"ascendio"并没有那么好的运气，第二次返回了nil。(魔女并不是总成功的)



工厂模式是一种老式的编程方法，在Swift中有更佳的实现方法——使用可失败构造器来实现。

删除create(_words:)方法，使用如下代码替换它:

```
init?(words: String) {
  if let incantation = MagicWords(rawValue: words) {
    self.magicWords = incantation
  }
  else {
    return nil
  }
}
```

在这里简化了代码，代码中未具体指明创建与返回Spell对象。

这几行现在出现了编译错误:

```
let first = Spell.create(withMagicWords: "abracadabra")
let second = Spell.create(withMagicWords: "ascendio")
```

需要改成新的构造器进行创建，使用如下代码替换上面的代码:

```
let first = Spell(words: "abracadabra")
let second = Spell(words: "ascendio")
```

这样一来就没有错误了，playground顺利编译通过。这样一改代码就更加简介了，不过还可以做的更好！:]

Guard语句

guard是一个用来断言某些事情为true的快捷方法：比如说，判断一个值是否大于0或一个条件值是否可以被解绑，在验证失败的情况下可以执行一段代码。

guard语句是在Swift 2中提出的，一般是通过调用堆栈进行冒泡错误处理，错误在最后才会被处理。Guard语句允许从函数或方法中较早的退出，这样就比需要判断条件满足之后的逻辑才会执行的方法更加清晰，简明。

将可失败构造器改为如下使用guard语句的形式来初始化:

```
init?(words: String) {
  guard let incantation = MagicWords(rawValue: words) else {
    return nil
  }
  self.magicWords = incantation
}
```

这样修改之后就不需要为了判断另一种情况额外增加一行代码了，使在失败的情况下的处理更加显眼，因为它放在了构造器的最前端。并且“黄金路径”的缩进最少。“黄金路径”是指当每件事都如预期即没有错误发生时的执行路径。更少的缩进更易于阅读。

注意到第一个和第二个咒语的常量值并未改变，而代码则变得更加精练了。

自定义处理避免错误

至此我们已经整理了Spell构造器的代码并且使用nil避免了一些错误，接下来学习如何进行更加复杂的错误处理。

进行这一节的学习请先打开Avoiding Errors with Custom Handling – Starter.playground。

看看下面的代码，观察它的特点:

```
struct Spell {
  var magicWords: MagicWords = .abracadbra

  init?(words: String) {
    guard let incantation = MagicWords(rawValue: words) else {
      return nil
    }
    self.magicWords = incantation
  }

  init?(magicWords: MagicWords) {
    self.magicWords = magicWords
  }
}
```

这是在教程第一部分中修改后的Spell的构造器。注意下面的Avatar协议和一个为了方便起见使用的可失败构造器，

```
protocol Familiar: Avatar {
  var noise: String { get }
  var name: String? { get set }
  init(name: String?)
}
```

在playground的后面会有一些动物(比如蝙蝠和蟾蜍)实现这个Familiar协议。

> 注意: 这里的familiar并不是熟悉的意思，指的是魔女或者魔法师的宠物或仆从，通常具有类人的能力，想象一下哈利波特中的Hedwig和绿野仙踪中的飞猴。
>
> 
>
> 这明显不是Hedwig(哈利波特中作为信使的宠物)，不过也挺可爱的，不是吗?

```
struct Witch: Magical {
  var avatar = "*"
  var name: String?
  var familiar: Familiar?
  var spells: [Spell] = []
  var hat: Hat?

  init(name: String?, familiar: Familiar?) {
    self.name = name
    self.familiar = familiar

    if let s = Spell(magicWords: .prestoChango) {
      self.spells = [s]
    }
  }

  init(name: String?, familiar: Familiar?, hat: Hat?) {
    self.init(name: name, familiar: familiar)
    self.hat = hat
  }

  func turnFamiliarIntoToad() -> Toad {
    if let hat = hat {
      if hat.isMagical { // 你见过魔女不带帽子就施放魔法吗 :]
        if let familiar = familiar {   // 检查魔女是否拥有仆从
          if let toad = familiar as? Toad {  // 若仆从是蟾蜍，则不需要魔法
            return toad
          } else {
            if hasSpell(ofType: .prestoChango) {
              if let name = familiar.name {
                return Toad(name: name)
              }
            }
          }
        }
      }
    }
    return Toad(name: "New Toad")  // 这是一只新的蟾蜍
  }

  func hasSpell(ofType type: MagicWords) -> Bool { // 检查魔女的魔法书中是否拥有某个咒语
    let change = spells.flatMap { spell in
      spell.magicWords == type
    }
    return change.count > 0
  }
}
```

最后是魔女，代码中会看到:

- 一个魔女是使用姓名和仆从或姓名、仆从和帽子初始化的。

- 一个魔女了解有限数量的咒语，并保存在Spell对象数组中。
- 一个魔女倾向于使用.prestoChango咒语将仆从变为蟾蜍，在turnFamiliarIntoToad()方法中。

应该注意到了turnFamiliarIntoToad()方法的长度和缩进的数量。若在函数中出错的话会返回一只全新的蟾蜍，对这个特定的咒语来说是一个令人困惑(也是错误的！)的输出结果，接下来会在下一节中使用自定义的错误处理来修改代码。

使用Swift错误进行重构

> “Swift提供了优秀的对运行时的错误抛出、捕获、传递和可恢复操作的支持。” -----The Swift Programming Language (Swift 3)

不要与末日神殿相混淆，Swifr中的金字塔是一种反模式的结构，需要使用多层嵌套的语句来实现控制流。在上面的turnFamiliarIntoToad()方法中可以看到 - 注意需要使用六个花括号才能闭合语句，要读懂这种多层嵌套的代码需要下一些功夫。



之前见过的guard语句与多个并列的可选绑定类型可以帮助简化金字塔型的代码，不过使用do-cathch的话可以通过在处理错误状态时进行控制流去耦合以解决这个问题。

do-catch机制通常伴随着下面这些关键字:

- throws
- do
- catch
- try
- defer
- Error

在实际操作do-catch时会抛出不同的自定义错误。首先，需要定义期望处理的状态，使用枚举列出可能发生错误的情况。

在Witch的定义前面添加如下代码:

```
enum ChangoSpellError: Error {
  case hatMissingOrNotMagical
  case noFamiliar
  case familiarAlreadyAToad
  case spellFailed(reason: String)
  case spellNotKnownToWitch
}
```

注意关于ChangoSpellError的两件事:

- 它符合Error协议，在Swift中定义错误时的必要条件。
- 在spellFailed的情况下可以自行通过一个关联值设置咒语失败时的原因。

> 注意: ChangoSpellError的命名是在咒语“Presto Chango!”施放之后 —— 当一个魔女将一个仆从转化为蟾蜍时所使用的魔法。

好了，准备好施放一些魔法了，在方法名中添加throws来标识调用这个方法后可能产生的错误:

```
func turnFamiliarIntoToad() throws -> Toad {
```

在Magical协议中更新下这个方法名:

```
protocol Magical: Avatar {
  var name: String? { get set }
  var spells: [Spell] { get set }
  func turnFamiliarIntoToad() throws -> Toad
}
```

现在将错误都用枚举列出来了，需要重新编写turnFamiliarIntoToad()方法，分开处理每种错误情况。

处理魔法帽错误

首先，修改如下语句以确保魔女带着那顶至关重要的帽子:

```
if let hat = hat {
```

改成下面这样:

```
guard let hat = hat else {
  throw ChangoSpellError.hatMissingOrNotMagical
}
```

> 注意: 不要忘了移除底部关联的花括号，否则会编译错误。

下面这行是帽子属性的布尔值验证:

```
if hat.isMagical {
```

应当使用额外的guard语句来执行这个验证，不过将验证语句打包在一起用一行搞定的话会更加简洁。将第一个guard语句如下整合在一起:

```
guard let hat = hat, hat.isMagical else {
  throw ChangoSpellError.hatMissingOrNotMagical
}
```

现在可以删掉if hat.isMagical { 验证了。

下一部分来拆分一下条件语句金字塔。

处理仆从错误

接下来检查魔女是否拥有仆从:

```
if let familiar = familiar {
```

修改成如下使用guard语句的形式，若没有仆从则抛出一个.noFamiliar错误:

```
guard let familiar = familiar else {
  throw ChangoSpellError.noFamiliar
}
```

目前先忽略其他情况下的错误，之后代码会进行处理。

处理蟾蜍错误

在之后的代码中若魔女调用turnFamiliarIntoToad()咒语则返回已存在的蟾蜍，这里最好有一个能指出她操作"失误"的error。修改下面的代码:

```
if let toad = familiar as? Toad {
  return toad
}
```

改成如下这样:

```
if familiar is Toad {
  throw ChangoSpellError.familiarAlreadyAToad
}
```

注意，把as?改为了is，这样简化了验证语句，没必要使用判断结果来判断是否符合协议。is关键句也常用于类型比较。

把else的相关语句都删掉，不需要了~

处理咒语错误

最后，调用hasSpell(_ type:)确保魔女吟唱的是正常的咒语，将下面的代码:

```
if hasSpell(ofType: .prestoChango) {
  if let name = familiar.name {
    return Toad(name: name)
  }
}
```

改成这样:

```
guard hasSpell(ofType: .prestoChango) else {
  throw ChangoSpellError.spellNotKnownToWitch
}

guard let name = familiar.name else {
  let reason = "Familiar doesn’t have a name."
  throw ChangoSpellError.spellFailed(reason: reason)
}

return Toad(name: name)
```

现在可以移除最后这行了:

```
return Toad(name: "New Toad")
```

这个方法变得更加简洁了，可以使用了。我在下面的代码中添加了一些注释，可以稍微解释下:

```
func turnFamiliarIntoToad() throws -> Toad {

  // When have you ever seen a Witch perform a spell without her magical hat on ? :]
  guard let hat = hat, hat.isMagical else {
    throw ChangoSpellError.hatMissingOrNotMagical
  }

  // 检查魔女是否拥有仆从
  guard let familiar = familiar else {
    throw ChangoSpellError.noFamiliar
  }

  // 检查仆从是否为蟾蜍 - 若是的话为何多此一举？
  if familiar is Toad {
    throw ChangoSpellError.familiarAlreadyAToad
  }
  guard hasSpell(ofType: .prestoChango) else {
    throw ChangoSpellError.spellNotKnownToWitch
  }

  // 检查仆从是否拥有名称
  guard let name = familiar.name else {
    let reason = "Familiar doesn’t have a name."
    throw ChangoSpellError.spellFailed(reason: reason)
  }

  // 检查完毕后返回一个与魔女仆从同名的蟾蜍
  return Toad(name: name)
}
```

你可以从turnFamiliarIntoToad()中返回一个可选类型来指示出“咒语吟唱时出错了”，不过使用自定义错误的话可以更明确地指出错误状态并作出对应的操作。

使用自定义错误的其他好处

现在你拥有了一个可以抛出自定义Swift错误的方法，接下来要处理它们，使用的这种机制一般被称为do-catch语句，类似其他语言比如Java中的try-catch机制。

在playground的底部添加如下代码:

```
func exampleOne() {
  print("") // 在调试区域添加一个空行

  // 1
  let salem = Cat(name: "Salem Saberhagen")
  salem.speak()

  // 2
  let witchOne = Witch(name: "Sabrina", familiar: salem)
  do {
    // 3
    try witchOne.turnFamiliarIntoToad()
  }
  // 4
  catch let error as ChangoSpellError {
    handle(spellError: error)
  }
  // 5
  catch {
    print("Something went wrong, are you feeling OK?")
  }
}
```

函数做了下面几件事:

1. 创建魔女的仆从，一只叫Salem的猫。
2. 创建魔女Sabrina。
3. 尝试把猫转换为蟾蜍。
4. 捕获到ChangoSpellError错误并适当地处理错误。
5. 最后捕获其它错误并打印一条信息。

添加完上述代码并运行后会看到一个编译错误 - 接着处理一下。

handle(spellError:)方法还没有定义，在exampleOne()函数的上方添加如下代码:

```
func handle(spellError error: ChangoSpellError) {
  let prefix = "Spell Failed."
  switch error {
    case .hatMissingOrNotMagical:
      print("\(prefix) Did you forget your hat, or does it need its batteries charged?"
)

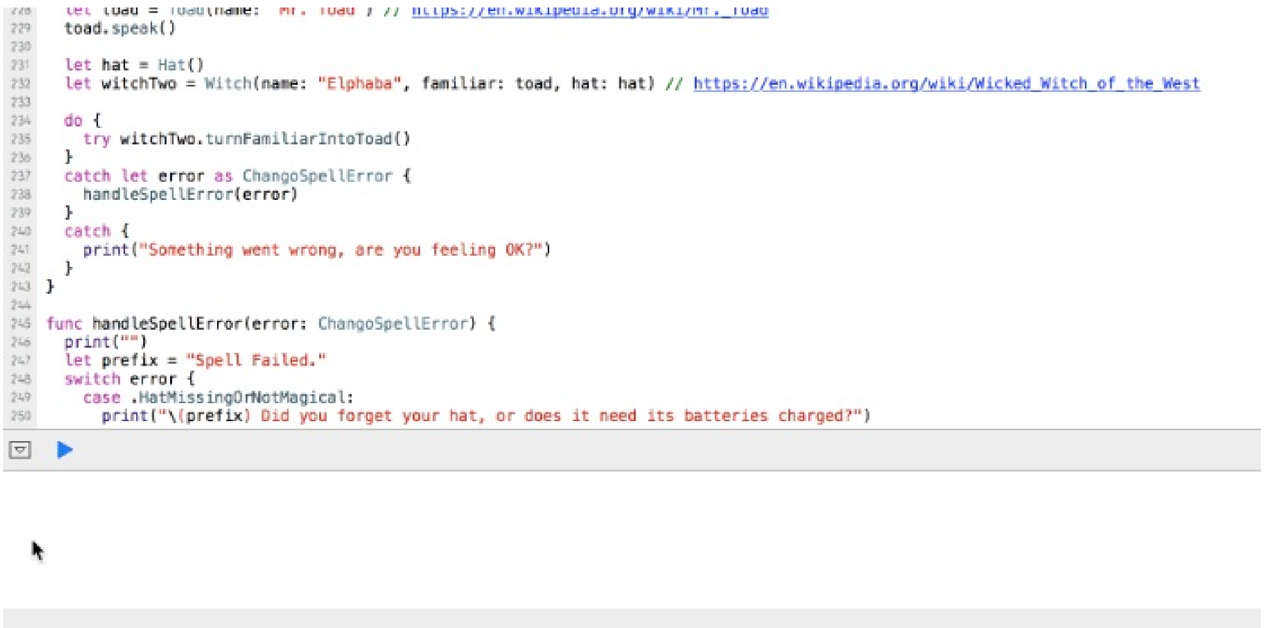
    case .familiarAlreadyAToad:
      print("\(prefix) Why are you trying to change a Toad into a Toad?")

    default:
      print(prefix)
  }
}
```

最后在playground的地步添加如下代码调用并运行:

```
exampleOne()
```

点击Xcode工作区左下角的箭头图标展开调试控制台，就可以看到playground的输出了:



错误捕获

下面是一些在上述代码片段中所出现的语言特性的简要介绍。

`catch`

可以在Swift中使用模式匹配处理特定错误或其它错误类型的主题。

上面的代码中使用了catch: 一个捕获特定的ChangoSpell错误，另外一个处理其余的错误情况。

`try`

使用try结合do-catch机制明确地指出哪一行或哪一块代码会抛出错误。

可以用如下几种方式使用try:

- try: 标准用法，也是上面所用到的。
- try?: 忽略要处理的错误，若抛出错误则返回nil。
- try!: 类似强制解绑，try!可用于加载某些确定存在的媒体文件，需谨慎使用。

是时候使用try?语句了，将下面的代码剪切并粘贴到playground的底部:

```
func exampleTwo() {
  print("") // 在调试区域添加一个空行

  let toad = Toad(name: "Mr. Toad")
  toad.speak()

  let hat = Hat()
  let witchTwo = Witch(name: "Elphaba", familiar: toad, hat: hat)

  print("") // 在调试区域添加一个空行

  let newToad = try? witchTwo.turnFamiliarIntoToad()
  if newToad != nil { // Same logic as: if let _ = newToad
    print("Successfully changed familiar into toad.")
  }
  else {
    print("Spell failed.")
  }
}
```

注意与exampleOne的不同，在这里并不关心发生的错误类型，不过当错误发生时还是会捕获。由于未创建蟾蜍，因此newToad的值是nil。

错误传递

`throws`

在Swift中，若一个函数或方法抛出错误时需要使用throw关键字，抛出的错误会自动传递给调用的堆栈，不过让错误冒泡的距离离源头太远通常被认为是不好的。代码库中显著的错误传递会增加错误未被有效处理的可能性，throws是对代码中记录传递过程的授权 - 给编程者留下传递的记录，以便有效的处理错误。

`rethrows`

至此为此看到的所有例子都用的是throws，没见过它的兄弟rethrows吧

rethrows告诉编译器仅当它的函数参数抛出错误时才抛出错误，下面是个简单的例子(不需要添加到playground中):

```
func doSomethingMagical(magicalOperation: () throws -> MagicalResult) rethrows -> Magi
calResult {
  return try magicalOperation()
}
```

这里的doSomethingMagical(_:)会在magicalOperation给函数抛出一个错误的情况下抛出错误，若是则返回一个MagicalResult。

操作错误处理的行为

`defer`

虽然让错误自动传递在大多数情况下是适用的，不过也有一些时候想控制错误在调用堆栈中传递时 app 的行为。

defer语句是一种跳出当前域时执行"清空"操作的机制，比如在一个方法或函数返回时。它有助于清理某些资源，不管操作会不会成功，这在处理错误上下文时很有效。

来实践一下，给Witch添加如下方法:

```
func speak() {
  defer {
    print("*cackles*")
  }
  print("Hello my pretties.")
}
```

在playground底部添加如下代码:

```
func exampleThree() {
  print("") // Add an empty line in the debug area

  let witchThree = Witch(name: "Hermione", familiar: nil, hat: nil)
  witchThree.speak()
}

exampleThree()
```

在调试控制台中，会看到魔女在说完话后会咯咯地笑(先输出"Hello my pretties"然后输出"*cackles*")。

有趣的是，defer语句会以相反的顺序输出我们缩写的print代码内容。

在speak()中添加第二个defer语句，这样魔女会在说完话后先尖叫(screeches)后咯咯地笑(cackles)。

```
func speak() {
  defer {
    print("*cackles*")
  }

  defer {
    print("*screeches*")
  }

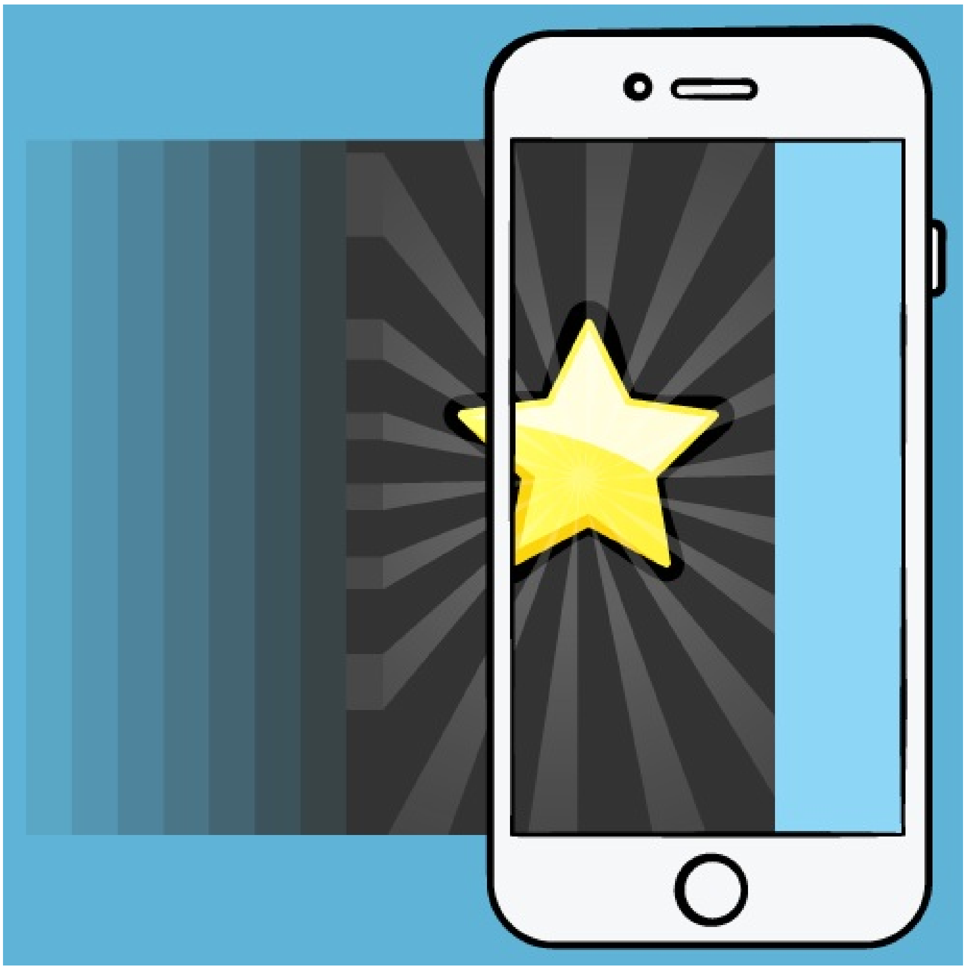
  print("Hello my pretties.")
}
```

这些语句如期按顺序输出了吗?是的，这就是defer的魅力!

UIPresentationController Tutorial: Getting Started

UIPresentationController指南:入门

> - 原文链接：UIPresentationController Tutorial: Getting Started
> - 原文作者：Ron Kliffer
> - 译者：yrq110



> 学习如何使用UIPresentationController进行自定义跳转。

在iOS的早些时期，视图控制器的跳转一直是iOS开发者工具箱中的一个独立模块，你可能以前使用过present(_:animated:completion:)方法，可能像其他大多数开发者一样使用iOS自带的跳转样式。在这篇UIPresentationController的教程中你会学习如何设置自定义的视图控制

器跳转样式，不再局限于全屏的、弹出式的默认跳转动画，将枯燥、无趣的视图跳转表现为更加的生动、有活力。

看完这篇教程后你会学到：

1. 如何创建一个UIPresentationController子类
2. 如何使用UIPresentationController进行圆滑的自定义跳转
3. 如何复用UIPresentationController，根据不同的controller调整参数
4. 如何创建自适应的转场动画支持不同设备的屏幕尺寸

> 前提: 这篇教程需要Xcode 8.0或更高版本，这是由于需要使用新版本的Swift语句，并且假定你熟悉Swift与iOS SDK。

入门

场景: 2016夏季奥运会结束后，一位客户想让你做一款app，统计国际体育赛事中不同运动员与国家的所得奖牌。

由于功能需求比较简单，你的赞助者要求使用一种比较cool的跳转方式来展示运动项目列表。

起初你觉得有点慌，之后发现只需要动动指头写一个跳转工具即可，小case!

下载开始工程并打开它。

浏览项目目录，熟悉一下以下元素:

- MainViewController.swift: 项目的主视图控制器，在之后的项目中仅需修改这个文件。the main controller of this project from which all presentations start. This will be the only existing file in the project that you'll modify.
- GamesTableViewController.swift: 显示一个用户可选择的奥运会列表
- MedalCountViewController.swift: 显示所选择奥运会的奖牌榜
- GamesDataStore.swift: 负责项目中的model层，创建并存储model对象

在开始之前，先构建并运行app看看它能做什么，会看到如下界面:



首先点击Summer按钮显示GamesTableViewController夏季赛事的菜单。

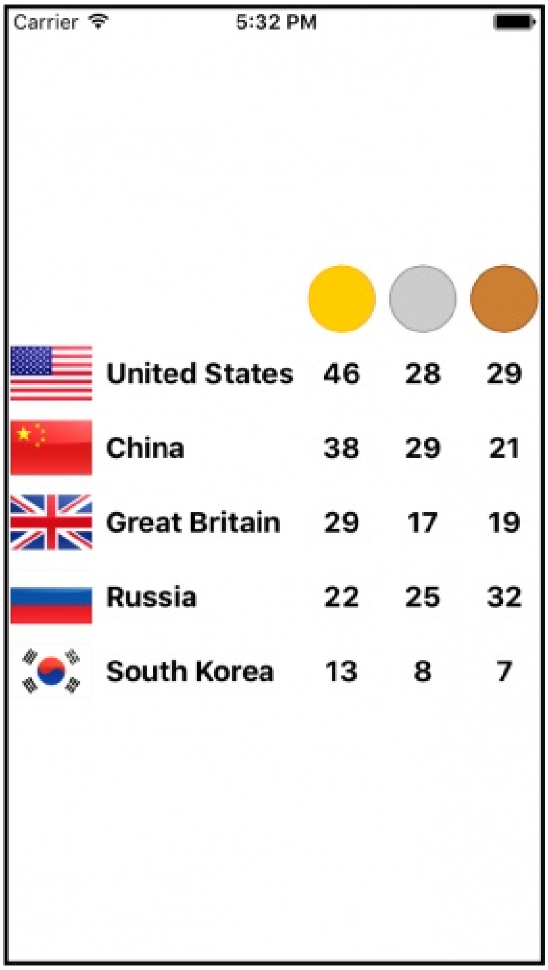


注意这个菜单是以默认方式展现的，而客户想要一个平滑滑入的效果。

接着选择London 2012奥运会回到主界面，看到一个更新后的logo。



最后点击Medal Count按钮跳转到MedalCountViewController，看到2012年奥运会的奖牌榜。



如你所见，这个控制器的跳转也是老式的自下而上弹出方式，点击屏幕隐藏它。

这个app就是需要你来修改的，是时候注意下有关UIPresentationController的核心概念与理论了。

iOS Transition的核心概念

When you call present(_:animated:completion:), iOS does two things. First, it instantiates a UIPresentationController. Second, it attaches the presented view controller to itself then presents it using one of the built-in modal presentation styles. You have the power to override this mechanism and provide your own UIPresentationController subclass for a custom presentation. Understanding these key components is mandatory if you want to build sweet presentations in your apps:

1. The presented view controller has a transitioning delegate that's responsible for loading the UIPresentationController and the presentation/dismissal animation controllers. That delegate is an object that conforms to UIViewControllerTransitioningDelegate.
2. The UIPresentationController subclass is an object that has many presentation-customizing methods, as you'll see later in the tutorial.
3. The animation controller object is responsible for the presentation and dismissal

animations. It conforms to UIViewControllerAnimatedTransitioning. Note that some use cases warrant two controllers: one for presentation and one for dismissal.

4. A presentation controller's delegate tells the presentation controller what to do when its trait collection changes. For the sake of adaptivity, the delegate must be an object that conforms to UIAdaptivePresentationControllerDelegate. That's all you need to know before you dive in!